HKU CS Tech Report TR-2015-03

Efficient Top-k Joins on Complex Data Types

Shuyao Qi
The University of Hong Kong, China
qisy@connect.hku.hk

Panagiotis Bouros
Aarhus University, Denmark
pbour@cs.au.dk

Nikos Mamoulis
The University of Hong Kong, China
nikos@cs.hku.hk

December 1, 2015

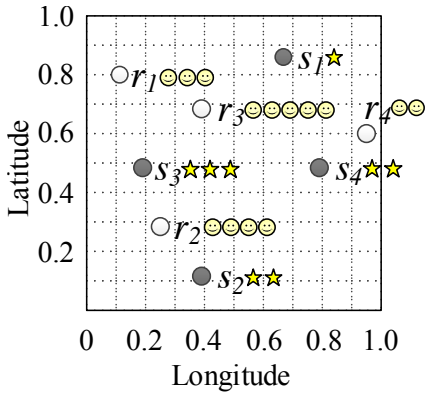
Abstract

Consider two collections of objects R and S, where each object is assigned a score (e.g., a rating). Given a join predicate ϕ and an integer k, a top-k join query returns the k pairs of objects which have the highest combined score (based on an aggregate scoring function γ) among all object pairs in $R \times S$ that qualify ϕ. This query type has been extensively studied in the relational database context where the join predicate is equality, with the main goal of minimizing the number of tuples accessed from relations R and S. However, if the top-k join involves a non-equijoin predicate ϕ on complex data types, the computational cost can easily become the bottleneck of query evaluation. In view of this, we propose a novel evaluation paradigm for top-k joins, which aims at minimizing the computations cost, without compromising the access cost. The main idea behind our paradigm is to examine blocks of data from R and S ordered by the object scores; by performing the top-k join in a block-wise fashion, we avoid (i) building expensive indexes incrementally and (ii) comparing pairs of blocks that may not contain results (using appropriate bounds). We show how our paradigm can be applied for the cases of top-k spatial and string joins and conduct an analysis on how to derive the optimal block size for each case. Finally, we evaluate our proposal by extensive experimentation on both real and synthetic data.

1 Introduction

Consider two collections of objects R and S, and assume that the objects in either R or S have (at least) one *join attribute* `att` and a *scoring attribute* `score`. Given a *predicate* ϕ (e.g., equality $=$) on the join attributes of two objects and a *monotone aggregation function* γ (e.g., SUM) which combines their scoring attributes, k-Join retrieves a k-subset J of $R \times S$ such that for every pair of objects $(r, s) \in J$, $\phi(r, s)$ is satisfied and for any $(r', s') \in R \times S \setminus J$ which satisfies $\phi(r', s')$, $\gamma(r, s) \geq \gamma(r', s')$ holds. Top-k join queries have been extensively studied for the case of equality join predicate on attributes of primitive data types such as numerical values [6, 12, 17, 20, 22, 23, 29, 33, 37]. In this paper, we study this problem for the case where the join predicate applies on complex data types.

Spatial Distance k-Join **(**k-SDJoin**).** Consider, for instance, the case of spatial locations as join attributes. A predicate ϕ could qualify pairs of objects (r, s) which are spatially close based on a *distance threshold* ϵ, i.e., $dist_{SD}(r, s) \leq \epsilon$ where $dist_{SD}(r, s)$ denotes the distance between the spatial locations of r and s. An exemplary application of k-SDJoin in this context is recommending to the visitors of a city the k pairs of restaurants and hotels within short distance from each other,



(a)

id	restaurant name	rating
r_1	New York Pancakes	3
r_2	La Bella Napoli	5
r_3	Luigi's Pizza	2
r_4	x-treme Burgers	1

id	restaurant name	rating
s_1	La Bella Napoli	5
s_2	Extreme Burgers	2
s_3	NY Pancakes	4
s_4	Louigi's Pizza	2

(b)

Figure 1: Example of (a) a k-SDJoin and (b) a k-SSJoin.

that have the top combined ratings. Figure 1(a) illustrates a collection R of four restaurants and a collection S of four hotels. The objects carry a score shown next to every point. Assuming that the qualifying pairs should have Euclidean distance at most $\epsilon = 0.3$ and $\gamma = SUM$, the result of 2-SDJoin contains pairs (r_2, s_3) with aggregate score 7 and (r_2, s_2) with aggregate score 6. k-SDJoin finds application also in emerging scientific fields like bioinformatics; for instance, the problem of identifying pairs of amino acids that exhibit "good" properties and therefore, contribute to the stability of a protein. The properties of amino acids like the "solvent accessibility" can be quantified as scores [25], and thus, the problem at hand can be modeled as a k-SDJoin query which identifies among pairs of amino acids that are close to each other with respect to their 3D location, the ones with the highest SUM of their "solvent accessibility" score.

String Similarity k-Join (k-SSJoin). As another example of a k-Join operator on complex data types consider the case of strings as join attributes. The string attribute value of an object may contain typographic errors or abbreviations. Thus, in the context of k-SSJoin, a predicate ϕ would qualify object pairs (r, s) with a similar textual description based on a *string distance threshold* ϵ, i.e., $dist_{SS}(r, s) \leq \epsilon$ where $dist_{SS}$ denotes the string distance (e.g., edit or Hamming distance) between the string attributes of r and s. k-SSJoin finds application in tasks like data integration where the score ratings of objects from different data sources are combined to identify the most dominant ones, or data cleaning and de-duplication. For instance, a person in search of a good restaurant would find value on a mashable-like search engine that ranks available options by combining ratings from different Web sources. Note that the users of a search engine usually iterate through the first part of the search results [14], i.e., the top-k restaurants, and therefore, a k-Join operator is more useful than computing the entire join between the sources. Under this scenario, Figure 1(b) illustrates two restaurant collections R and S from different websites which however may store the same object under a different name. Assuming that the qualifying pairs should have edit distance at most $\epsilon = 2$ and $\gamma = AVG$, the result of 2-SSJoin contains pairs (r_2, s_1) with aggregate score 5 and (r_3, s_4) with aggregate score 2.

Existing Solutions. A straightforward approach for the evaluation of the k-Join operator is to first compute the $R \bowtie_\phi S$ join with respect to predicate ϕ on the join attribute of the objects, and then, identify the top-k result pairs based on the aggregate score function γ. This *Join-First Paradigm* (JFP) is discussed in more detail in Section 3.2. If input collections R and S are accessed in decreasing order of their `score` values, k-Join is evaluated by accessing R and S only partially, using bounds for the non-examined objects to terminate [12, 22, 29]. Specifically, the *Score-First Paradigm* (SFP) examines the objects from R and S incrementally in decreasing order of their `score` values. Each time, an object r (or s) is accessed from R (or S) and joined with the (buffered) objects of S (or R), which have been previously accessed. The buffered objects from S (or R) are indexed by $\mathcal{I}_S$ (or $\mathcal{I}_R$) on join attribute `att`; for instance in case of relational top-k equijoins $\mathcal{I}_S$ (or $\mathcal{I}_R$) is a hash-table. After r (or s) has been examined, it is then buffered, i.e., inserted into $\mathcal{I}_R$ (or $\mathcal{I}_S$). Join results are organized in a priority queue based on their aggregate

score; all results that are guaranteed to be of higher aggregate score compared to the maximum possible score of join pairs not computed yet are incrementally produced. SFP is presented in detail in Section 3.1.

Motivation. Previous k-Join evaluation techniques [6, 12, 17, 20, 22, 29, 33, 37] have focused on applying SFP on relational top-k equijoins. Their primary goal has been the minimization of accesses from R and S, which corresponds to I/O cost in a centralized setting (assuming unlimited memory for buffering/indexing the accessed objects) [12, 17, 20, 22, 29] or to communication cost in the case where R and S (or parts thereof) reside in different nodes [6, 33, 37]. We observe that, in case of top-k equijoins, the computational cost of SFP is very low because the $\mathcal{I}_R$ and $\mathcal{I}_S$ indices are hash tables which support search and updates in constant time. However, as we show in Section 8.7, when `att` is an attribute of a complex type and ϕ is not equality (e.g., a spatial distance join or string similarity join), the computational cost of SFP can easily exceed the access cost, due to the increased overhead of searching and incrementally updating $\mathcal{I}_R$ and $\mathcal{I}_S$. For instance, in the case of a spatial distance k-Join, $\mathcal{I}_R$ and $\mathcal{I}_S$ should be spatial indices (e.g., R-trees) which are significantly more expensive to search and incrementally update compared to hash tables. Therefore, the focus of our work is the efficient (in terms of CPU cost) k-Join evaluation on complex data types.

Contribution. We propose a novel, hybrid evaluation paradigm, for the k-Join operator termed the *Block-based Paradigm* (BLP). BLP processes the objects in decreasing order of their scoring attribute similar to SFP but in a block-wise fashion, and joins blocks of the input collections in a similar to JFP manner while using score bounds to avoid computing the entire join of the input collections. As we discuss in detail in Section 2.2, BLP differs from previous block-based extensions of k-Join [4, 29]; for example, in [29] a simple extension to SFP that accesses one block of objects at a time is proposed, but the objects are processed one-by-one as in the original SFP algorithm. Note that despite focusing on binary k-Join under one scoring attribute per input, as we discuss in Section 4.3, BLP can directly incorporate the pulling strategy and the bound scheme from [8] for multiple scores, and handle multiple inputs either as a hierarchy of binary k-Join queries [17] or in a multi-way fashion [29].

The performance of BLP is strongly related to the size λ of the blocks accessed from the input collections. Under this, we introduce *objective cost function* $\mathcal{C}(\lambda)$ to capture the total cost of computing k-Join with BLP, and then model the selection of the most appropriate block size as an optimization problem. Further, we devise a novel model for estimating the number of objects accessed from each collection which, in contrast to previously proposed models [6, 13, 17, 30], employs cheap-to-compute statistics and does not require any prior assumptions regarding the distribution of the join or the scoring attributes. In fact, due to employing expensive statistics, i.e., multi-dimensional histograms, the models proposed in [6, 30] for estimating the number of accessed objects cannot be employed in cases other than equijoins on primitive numerical join attributes.

We study the use cases of *spatial distance* k-Join (k-SDJoin) and *string similarity* k-Join (k-SSJoin). We discuss in detail how SFP, JFP and BLP are applied to compute these novel query operations, employing special indexing structures and optimization techniques. Finally, we conduct an extensive experimental evaluation showing that BLP greatly outperforms SFP and JFP in the computation of k-SDJoin and k-SSJoin queries. In our experiments, we employ both real and synthetic score-carrying object collections. Our analysis demonstrates also the effectiveness of our model for estimating the number of objects needed to be accessed from each collection in order to compute a k-Join query, compared to the model proposed in [13] (note that in the absence of expensive statistics [30] assumes uniform and independent distribution similar to [13]), and the accuracy of our model for selecting the block size for BLP.

In a preliminary version of this paper [27], we presented BLP and demonstrated its efficiency for k-SDJoin. However, the problem of selecting the optimal block size λ was not studied and also, the application of BLP on other join types (e.g. k-SSJoin) was not addressed.

Outline. The rest of the paper is organized as follows. Section 2 reviews related work and Section 3 discusses in detail the existing evaluation paradigms for k-Join. Section 4 presents our novel paradigm BLP for efficient evaluation of the k-Join operator. Sections 5 and 6 discuss the

application of all three paradigms for spatial and string join attributes, respectively. Section 7 introduces our models for selecting block size and estimating the number of accessed objects. Comprehensive experiments and our findings are reported in Section 8. Finally, Section 9 concludes the paper and discusses directions for future work.

2 Related Work

Our work is related to top-k queries and joins. In addition, the use cases of k-SDJoin and k-SSJoin are related to spatial and string joins, respectively. Sections 2.1 to 2.5 summarize related work on these subjects.

2.1 Top-k Queries

Fagin et al. [7] present an analytical study of various methods for top-k aggregation of ranked inputs by monotone aggregate functions. Consider a collection of objects (e.g., restaurants) which have scores (i.e., rankings) at two or more different sources (e.g., different ranking websites). Given an aggregate function γ (e.g., SUM) the top-k query returns the k restaurants with the highest aggregated score (from the different sources). Each source is assumed to provide a sorted list of the objects according to their atomic score there; requests for random accesses of scores based on object identifiers may be also possible. For the case where both sorted and random accesses are possible, the *Threshold Algorithm* (TA) retrieves objects from the ranked inputs (e.g., in a round-robin fashion) and a priority queue is used to organize the best k objects seen so far. Let ℓ_i be the last score seen in source S_i; $T = \gamma(\ell_1, ..., \ell_m)$ defines a lower bound for the aggregate score of objects never seen in any S_i yet. If the k-th highest aggregate score found so far is no less than T, the algorithm is guaranteed to have found the top-k objects and terminates. For the case where only sorted accesses are possible, [20] presents an optimized implementation of the *No-Random accesses Algorithm* (NRA), originally proposed also in [7]. The top-k results are incrementally fetched based on their aggregate scores. [36] presents a framework for top-k queries on top of collections having multi-attribute indices. An *index-merge* paradigm is proposed to merge multiple index nodes progressively and selectively.

2.2 Top-k Joins

The top-k query is a special case of the general top-k join query, which performs rank aggregation on top of relational join results. Natsev et al. [22] are the first to study top-k join evaluation. The J* algorithm is a multi-way join operator, which takes as input two or more input streams, one per collection; in each stream, the objects of the corresponding collection are ranked based on their scores. Objects are accessed incrementally from the streams (e.g., in round-robin). Partial join results are computed and given upper bounds based on the maximum possible aggregate scores of complete join results that may include them. The algorithm maintains a heap for all partial and complete join results. At each step, the object combination at the top of the heap is popped, missing values are sought for it (if partial) by accessing the streams and the results are pushed back to the heap. J* incrementally outputs the top combinations in the heap if they are complete join results.

Ilyas et al. [12] propose a binary operator, called *Hash-based Rank-JoiN* (HRJN*) for top-k joins, which produces results incrementally and therefore can be used multiple times in a multi-way join evaluation plan. Assume that the objects of R and S are accessed incrementally based on the values of their `score` attribute. HRJN* accesses objects from R (or S) and joins them using the join attribute `att` with the buffered objects of S (or R), which have been previously accessed (these objects are buffered and indexed by a hash-table). Join results are organized in a priority queue based on their aggregate score. Let ℓ_R, h_R (ℓ_S, h_S) be the lowest and highest scores seen in R (S) so far; all join results currently in the queue having aggregate score larger than *threshold* $T = \max\{\gamma(h_R, \ell_S), \gamma(\ell_R, h_S)\}$ are guaranteed to have higher aggregate score than any join result not found so far and therefore can be output (or pipelined to the next operator). The follow-up work in [17] applies HRJN* in the more general problems of multiple input collections (similar to [22]) that contain one or more scoring attributes each. The optimality of algorithms for such

generalized top-k joins is studied in [29] by defining the *Pull/Bound Rank Join* (PBRJ) evaluation framework; in fact, HRJN* [12] is an instantiation of this framework, denoted by PBRJ$^*_c$, which applies the threshold-adaptive pulling strategy for accessing objects from the input collections and uses the so-called corner bound scheme for specifying threshold T, described above. In Section 3.1, we present a generalization of the binary HRJN*/PBRJ$^*_c$, i.e., the *Score-First Paradigm* (SFP), where any join predicate can be used (HRJN*/PBRJ$^*_c$ assume only equijoins) and each collection contains a single scoring attribute. The cases of multiple input collections and/or multiple scoring attributes per input are out of the scope of our work. In fact as Section 4.3 discusses, our proposed *Block-Based Paradigm* (BLP) could directly incorporate the methodology of [8] to optimize k-Join evaluation with multiple scoring attributes. On the other hand, existing database systems implement binary physical operators and thus, considering methodologies like [17, 29] for multiple inputs can be seen as an interesting direction for future work.

Further in [29], Schnaitter et al. shortly discuss a variant of the PBRJ framework where a block of objects is accessed at a time, instead of a single object; considering any type of join predicate ϕ a block-based variant of SFP is defined. Such a variant is proven to be instance optimal in terms of the number of objects accessed from each input collection. In Section 4.2 we build upon this analysis to establish that BLP is also instance optimal with respect to object accesses. Note however that, in practice the block-based variant of SFP significantly differs from our BLP proposal: although the objects are accessed in blocks they are still processed one-by-one as in the original SFP which means that every examined object is still probed against the entire set of buffered objects from the other collection. In contrast, the currently processed block in BLP is joined only with a small number of blocks of the other input collection. Chakrabarti et al. [4] also discuss a block-based evaluation strategy in the context of top-k keyword search where the join operator involves the intersection of compressed posting lists. The range of the document ids is first split into intervals and an upper aggregate score bound is defined for each interval. Then, the lists intersection is performed at the interval level while using score bounds to enable interval-based pruning. Compared to BLP, the work in [4] differs in two ways: (i) it primarily focuses on minimizing the decompression cost for the postings list and (ii) the interval-based partitioning and joining is strongly related to the problem at hand, i.e., keyword search, and cannot be applied to other type of join attributes and predicates.

Our work is also related to studies [6, 13, 17, 30] on estimating the *depth* of top-k join operators, i.e., the number of objects accessed from each input collection. Compared to our estimation methodology in Section 7.2, these studies either make specific assumptions regarding the objects of the collections or employ expensive statistics. Particularly, Ilyas et al. [13] propose a probability based model with the assumptions that the join and scoring attributes are (i) uniformly distributed, and (ii) independent from each other. Our experimental analysis in Section 8.4 shows that this model is prone to errors since the above assumptions are often not applicable on real data. Li et al. [17] propose a sampling-based approach where termination score θ is estimated performing a rank join process on uniformly sampled data. However, such a sampling method usually overestimates the depth, especially if both k (i.e., the number of required results) and the sampling ratio are small. Schnaitter et al. [30] address the above issues assuming that the distribution of the scoring and the join attribute is known in advance as a *frequency tensor* $F(r_i, s_j)$ which is then used to determine the number of join results with a specific aggregate score. To calculate F the authors employ multi-dimensional histograms [26]. Similar statistics (2-dimentional histograms) are considered by [6]. Yet, these approaches cannot be adopted for the problem studied in this paper as multi-dimensional histograms are efficient to compute and accurate only for join attributes of small domains and simple join predicates such as equality. Note that in the absence of such statistics [30] assumes uniform and independent distribution similar to [13].

Finally, in [6, 23, 33] a different aspect of the top-k join query was addressed; the case when the input collections or part of them originate from different physical locations. Wu et al. [33] model this problem utilizing a graph and a branch-and-bound algorithm is proposed that minimizes the number of network accesses required for computing the results of the top-k join query. In contrast, Doulkeridis et al. [6] determine the number of objects to be accessed from each network input, through the depth estimation procedure. Ntarmos et al [23] study top-k joins in NoSQL databases employing statistical structures (similar to 2-dimensional histograms) to reduce object accesses and effectively estimate the top-k join results in a distributed environment.

2.3 Top-k Joins and Complex Data Types

There exist very few works on top-k joins related to complex data types but the definition of the problems at hand significantly differ from the k-Join operator we study in our work. Specifically, the "top-k spatial join" over two collections R and S is defined in [38]; however, this query retrieves k objects in R intersecting the maximum number of objects from S. Thus, the ranking criterion is based on the number of spatial intersections and not on the aggregation of (non-spatial) scores from the two inputs as in case of the k-SDJoin realization of k-Join. Similarly, the "top-k similarity join" in [16, 35] is different from a specialization of k-Join as the ranking criterion is based on the set-similarity of the join attributes. Last, the "proximity rank join" over collections whose objects carry a feature vector and a scoring attribute, is defined in [21]. This query also differs from k-Join as (i) it additionally involves a query object and (ii) the ranking criterion is based on a combination of the scoring attributes, and the distance of the collection objects to each other and to the query object.

The only work to our knowledge, closely related to k-Join on spatial join attributes is [19], which studies a spatial join between two collections R and S containing objects associated with probabilistic values; each object o (e.g., a biological cell) is defined by a set of probabilistic locations and it is also assigned a confidence p_o to belong to a specific cell class. Given objects r from R and s from S, a score of the (r, s) pair is defined by multiplying their confidence probabilities p_r and p_s, and also considering distance $dist_{SD}(r, s)$ between their uncertain locations. Then, the top-k probabilistic join between R and S returns the top-k object pairs in order of their scores. Compared to k-SDJoin, the problem definition in [19] is different. The aggregate score function for k-SDJoin does not involve the distance of the objects, but the distance is used in the join predicate. Further, the solution proposed in [19] is of limited applicability as it is bound to a specific aggregation function and can efficiently work only with the L_1 distance.

2.4 Spatial Joins

Given two collections of spatial objects R and S the ϵ-*distance* join identifies the object pairs (r, s) with $r \in R$ and $s \in S$, such that $dist_{SD}(r, s) \leq \epsilon$, where $dist_{SD}(\cdot, \cdot)$ denotes the spatial (e.g., Euclidean) distance. An ϵ-distance join can be processed similarly to a *spatial intersection* join [3]. Specifically, assuming that each of collections R and S are indexed by an R-tree, the two R-trees are concurrently traversed by recursively visiting pairs of entries (e_R, e_S) for which their MBRs have minimum distance at most ϵ. Minimizing the cost of computing the distance between an MBR and an object was studied in [5]. For non-indexed inputs, alternative spatial join algorithms can be applied (e.g., the algorithm of [1] based on external sorting and plane sweep). Finally, distance joins have been studied also for high-dimensional data like image feature vectors. In this case, grid-based solutions [2] are preferred due to the ineffectiveness of R-trees in high dimensional spaces.

2.5 String Joins

Given two collections of string objects, the string similarity join finds all similar object pairs with respect to a string similarity measure, e.g., the edit distance, and a threshold ϵ. Most of the existing solutions [10, 18, 28, 32, 34] employ a filter-and-refinement evaluation framework. Gravano et al. [10] propose string join techniques on top of commercial databases. By matching *q-grams* and taking into account both positions and total number of matches, several pruning techniques are proposed pairs not within the desired edit distance. Xiao et al. propose ED-Join [34] which is also a q-gram-based method. However, it enhances the filter process by exploiting an edit distance lower bound derived from the location-based and content-based q-grams *mismatching*. Trie-Join [32] is a trie index based framework which processes string joins using prefix filtering to generate similar string pairs without the refinement step. Yet, Trie-Join is only efficient for short strings. The study in [31] showed that Pass-Join [18] is the most efficient method for both short and long strings. In Section 6.1, we discuss Pass-Join in detail and show how it can be integrated into k-SSJoin evaluation paradigms.

notation	description
R/S	Input object collections
$\mathcal{I}_R/\mathcal{I}_S$	Indices on input object collections
k	The number of required results
ϕ	A join predicate
γ	A monotone aggregate function
C	Candidate/result set of k-Join
θ	k-th highest aggregate score, i.e., lowest score in C

Table 1: Notation

3 Background: Existing Evaluation Paradigms

According to the definition of k-Join in Section 1, the result consists of object pairs with (i) join attributes qualifying a predicate ϕ, and (ii) high aggregate score based on a monotone aggregate function γ. In other words, k-Join is a combination of a join and a top-k query. Under this, we discuss two evaluation paradigms based on existing literature that prioritize either of the components of k-Join; Sections 3.1 and 3.2 primarily consider the scoring and the join attributes in the k-Join evaluation, respectively. Table 1 summarizes the notation used in this section and in the rest of the paper.

3.1 The Score-First Paradigm

The *Score-First Paradigm* (SFP) builds upon the methods proposed in [12, 29]. SFP is in fact a generalization of binary HRJN*/PBRJ$^*_c$ that works with any type of join attributes and predicates. Similar to HRJN*, SFP presumes that both input collections R and S are sorted on the scoring attribute of their objects in decreasing order. This can be the case if they stem from underlying operators which produce such interesting orders; otherwise R and S need to be sorted. In principle, SFP incrementally accesses objects from either R or S and joins them with the objects already examined from S or R, respectively. To determine which collection to access an object from, SFP keeps track of the last seen score ℓ_R from R and ℓ_S from S. In addition, to facilitate the join procedure, it maintains two indexing structures denoted by $\mathcal{I}_R$ and $\mathcal{I}_S$, which organize the buffered objects accessed so far.[1] Finally, SFP maintains set C of join pairs already found with the k highest aggregate scores, organized as a min-heap, and uses the lowest score θ in C as a bound for pruning and termination.

Paradigm 1 is a high-level pseudo code of the Score-First Paradigm. SFP receives as input two collections of objects R and S, a predicate ϕ on their join attributes `att`, a monotone aggregate function γ on their scoring attributes `score`, and an integer k. First, in Lines 1–4, SFP sorts (if needed) inputs R and S, and initializes indices $\mathcal{I}_R$ and $\mathcal{I}_S$, min-heap C and bound θ. Further, it initializes last seen scores ℓ_R and ℓ_S to ∞ in Line 5. Next, in Lines 6–15, it incrementally accesses objects from collection R or S and evaluates the k-Join query. On each iteration, the paradigm first decides which collection should be accessed and consequently, which object will be examined. Following the pulling strategy of HRJN*, SFP reads the next object from the collection with the highest last seen score, i.e., highest between ℓ_R and ℓ_S. Without loss of generality, assume that the next object to be examined is r accessed from R (i.e., $i = R$, $j = S$ and $o_i = r$ in Lines 7, 8 and 9, respectively); the other case is symmetric. Then, with current object r, SFP performs the following steps:

(i) It updates the termination threshold $T = \max\{\gamma(h_R, \ell_S), \gamma(\ell_R, h_S)\}$ following the corner bounding scheme of HRJN* in Line 11. From Section 2.2, recall that h_R and h_S are the highest scores seen in R and S, respectively, i.e., they equal the score of the very first object in each input. Note that the examination order of the objects employed by SFP allows threshold T to decrease faster and hence, SFP to terminate earlier.

(ii) It probes r against index $\mathcal{I}_S$ to retrieve objects $s \in S$, such that pair (r, s) qualifies predicate ϕ and $\gamma(r, s) > \theta$. To this end, SFP invokes the Probe procedure in Line 12. Probe employs $\mathcal{I}_S$

[1] In the case of a top-k equijoin, $\mathcal{I}_R$ and $\mathcal{I}_S$ are hash-tables as discussed in Section 2.2.

PARADIGM 1: Score-First Paradigm (SFP)

```
   Input  : R, S, φ, γ, k
   Output: C
1  sort R and S in descending order of the score attribute, if not already sorted;
2  initialize a min-heap C := ∅ of candidate results;
3  initialize θ := −∞;
4  initialize indices I_R := ∅ and I_S := ∅;
5  initialize ℓ_R := ∞ and ℓ_S := ∞;
6  while more objects exist in R and S do
7      i := next collection to be accessed ;                    // ℓ_S > ℓ_R ? S : R
8      j := the other collection;
9      o_i := get_next(i) ;                          // get next object from collection i
10     ℓ_i := score of o_i ;                  // update last seen score from collection i
11     T := max{γ(h_R, ℓ_S), γ(ℓ_R, h_S)} ;                          // HRJN* threshold
12     ⟨θ, C⟩ := Probe(o_i, I_j, T, φ, γ, k, θ, C);
13     if T ≤ θ then
14         break ;                          // result secured; no need to access more
15     insert o_i to I_i;
16 return C;
```

to identify every qualifying pair (r, s) and immediately after, it updates C and θ as follows. If $|C| < k$, pair (r, s) is inserted into candidates set C regardless of its aggregate score. Otherwise, (r, s) is inserted into C only if $\gamma(r, s) > \theta$ and in this case it *replaces* the k-th pair in C, such that set C always keeps the best k pairs found so far. Finally, in either case, θ is updated to the k-th aggregate score in C.

(iii) It checks if the evaluation of the k-Join query can terminate in Line 13. Specifically, as soon as $T \leq \theta$, SFP terminates reporting C as the query result.

(iv) It updates index $\mathcal{I}_R$ on R by inserting object r in Line 15 which will be probed by objects in S in future iterations.

3.2 The Join-First Paradigm

The *Join-First Paradigm* (JFP) prioritizes the join component of the k-Join query; JFP first joins the input collections R and S to get the object pairs that qualify predicate ϕ, and then retrieves the top-k pairs among them with the highest aggregate score. Paradigm 2 is a high-level pseudo code of the Join-First Paradigm. Similar to SFP, JFP receives as input collections R and S, a join predicate ϕ, a monotone aggregate score function γ, and an integer k. Note, though, that JFP does not make any pre-assumptions regarding the order of the objects in R and S. JFP also employs a min-heap C of size k to produce the final results which is initialized in Line 1. Then, in Lines 2–4, the paradigm computes the join between collections R and S based on predicate ϕ invoking the Join procedure. Join passes each join result to the heap C, which keeps track of the k pairs with the highest aggregate scores. The type of the join attributes and the nature of the predicate ϕ determine the strategy for performing the actual join. Without loss of generality, we assume at this point that the Join procedure performs an index join between $\mathcal{I}_R$ and $\mathcal{I}_S$ created for collection R and S in Lines 2–3, respectively.

Depending on the type of join attributes and predicate, the performance of JFP can be enhanced, e.g., by the use of special indexing schemes or by sorting the input collections. The idea behind this is to avoid computing the entire $R \bowtie_\phi S$ join. For instance, in Sections 5.2 and 6.3, we show how to use appropriate data structures in order to allow JFP to examine candidate result pairs in decreasing order of their aggregate scores, and thus, avoid joining the entire object collections.

PARADIGM 2: Join-First Paradigm (JFP)

Input : R, S, ϕ, γ, k
Output: C
1 initialize a min-heap $C := \emptyset$ of candidate results;
2 $\mathcal{I}_R := create\_index(R)$;
3 $\mathcal{I}_S := create\_index(S)$;
4 $C :=$ Join$(\mathcal{I}_R, \mathcal{I}_S, \phi, \gamma, k)$;
5 **return** C;

4 The Block-Based Paradigm

So far, we discussed how existing work is employed to compute k-Join, i.e., the Score-First and the Join-First Paradigms. However, due to primarily focusing on either the top-k or the join component of a k-Join query, both paradigms have certain shortcomings. Specifically, SFP is expected to be fast only if the k-Join results are found after a few accesses over sorted collections R and S. In contrast, if the best pairs include objects deep in the collections, the overhead of repeatedly updating and probing the $\mathcal{I}_R$ and $\mathcal{I}_S$ indices will be high which will also slow down the evaluation of the k-Join query. On the other hand, JFP is expected to be slower than SFP especially over large inputs and small values of k because of computing the entire join between the input collections. Even if special index structures are used to avoid computing the entire join, JFP may still need to index the entire input collections although part of them do not contribute to the k-Join result.

In this section, we propose a novel evaluation paradigm for the k-Join operator called the *Block-based Paradigm* (BLP) that alleviates the aforementioned shortcomings of SFP and JFP. BLP, like SFP, processes the objects in decreasing order of their scoring attributes but in a block-wise fashion, and joins blocks of the input collections using score bounds to early terminate the k-Join evaluation. Section 4.1 presents in detail the Block-based Paradigm. Then, Section 4.2 establishes the optimality of BLP in terms of the number of accessed objects while Section 4.3 discusses the extension of our work in case of multiple scoring attributes.

4.1 Description

Similar to SFP, BLP examines the objects in decreasing order of their scores. However, instead of probing *each* accessed *object* against buffered objects of the other collection seen so far, BLP each time probes a *block* of objects against the buffered *blocks* of objects from the other collection. Moreover, before probing a new block of objects, BLP creates an index for this block, and thus, the block-level probe can be performed by running instances of JFP. Under this perspective, BLP can be considered as an adaptation of JFP at the block level which, however, avoids to compute the entire $R \bowtie_\phi S$. For this purpose, BLP associates each accessed block of objects b with a lower score bound b^ℓ and an upper score bound b^u. Since, the objects inside b are in decreasing order of their scoring attributes, b^u (b^ℓ) corresponds to the score of the first (last) object inside b.

Paradigm 3 is a high-level pseudo code of the Block-based Paradigm. Similar to SFP and JFP, BLP receives as input two collections R and S, a join predicate ϕ, a monotone aggregate score function γ, and an integer k. Initially, in Lines 1–3, BLP first sorts (if needed) the input collections R and S (similar to SFP) and it initializes the min-heap C for the candidate result pairs and the θ bound. In addition, similar to SFP, it initializes the last seen scores ℓ_R and ℓ_S from R and S in Line 4. Next, in Lines 5–16, BLP evaluates the k-Join query following an approach similar to SFP, but it examines one block of objects, instead of one object, at a time. The procedure of selecting the input collection and the block to be accessed in Lines 6–9 is exactly the same as in SFP; note that ℓ_R and ℓ_S are updated to the score of the very last object inside the currently examined block (see Line 9). Without loss of generality assume that the next block b_i is to be accessed from collection R, i.e., $i = R$, $b_i = b_R$ and $j = S$; the other case is symmetric. BLP first constructs the $\mathcal{I}_{b_R}$ index for the current block b_R and then joins it with every block b_S accessed (and buffered) so far from collection S. The b_S blocks are considered in decreasing order of their score ranges (i.e.,

first b_{S_1}, then b_{S_2} etc). A pair of blocks b_R and b_S is joined in a similar fashion to a JFP call of the Join procedure with two important differences, though. First, BLP decides whether the join should take place, based on the following idea. Aggregate score $\gamma(b_R^u, b_S^u)$ represents an upper score bound for all object pairs in $b_R \bowtie_\phi b_S$. If we have already found at least k candidate pairs, i.e., $|C| = k$, then we know that joining b_R with b_S is pointless when $\gamma(b_R^u, b_S^u) \leq \theta$ (recall that θ is the k-th best aggregate score so far). In other words, the current block b_R from R is only joined with the blocks b_S from S for which $\gamma(b_R^u, b_S^u) > \theta$. Second, Join updates both the min-heap C and θ, similar to SFP. Finally, after handling current block b_R, BLP updates the termination threshold T and checks the termination condition in Lines 14–16. Note that threshold T is the same as in SFP with h_R and h_S being the highest score from R and S, respectively, i.e., they are equal to the score of the very first object inside b_{R_1} and b_{S_1}.

PARADIGM 3: Block-based Paradigm (BLP)

```
   Input  : R, S, φ, γ, k
   Output: C
 1 sort R and S in descending order of the score attribute, if not already sorted;
 2 initialize a min-heap C := ∅ of candidate results;
 3 initialize θ := −∞;
 4 initialize ℓ_R := ∞ and ℓ_S := ∞;
 5 while more blocks of objects exist in R and S do
 6     i := next collection to be accessed ;                    // ℓ_S > ℓ_R ? S : R
 7     j := the other collection;
 8     b_i := get_next_block(i, λ) ;        // get next block of objects from collection i
 9     ℓ_i := b_i^ℓ ;                          // update last seen score from collection i
10     I_{b_i} := create_index(b_i);
11     for each block b_j of j do
12         if γ(b_i^u, b_j^u) > θ then
13             ⟨θ, C⟩ := Join(I_{b_i}, I_{b_j}, T, φ, γ, k, θ, C);
14     T := max{γ(h_R, ℓ_S), γ(ℓ_R, h_S)};
15     if T ≤ θ then
16         break ;                          // result secured; no need to access more
17 return C
```

Despite the resemblance, BLP has two major advantages over SFP. First, performing the join at the block level is more efficient as (i) each block of objects is indexed just once and efficiently in a bulk-loading manner instead of iteratively inserting objects as in SFP, and (ii) the index of a given block can be used for multiple block-level joins. Second, in BLP, the currently processed block is joined only with a small number of blocks of the other input collection (with the help of the upper score bounds of the blocks), while in SFP the current object is probed against the entire set of objects buffered from the other input's collection. This makes a big difference if the performance of the index depends on its size (e.g., consider the difference between a spatial index with logarithmic search cost vs. a simple hash-table with constant look-up and update cost).

4.2 Instance Optimality

We analyze the performance of the Block-based Paradigm based on the notion of instance optimality defined by Fagin et al. [7]. We first introduce the necessary notation and discuss the connection to the study in [29].

Definition 4.1 (k-Join Instance) An instance of the k-Join problem is an (R, S, ϕ, γ, k) tuple such that input collections R and S are accessed in decreasing order of their score attribute, ϕ is a join predicate, γ is a monotone aggregate function, and $1 \leq k \leq |R \bowtie_\phi S|$.

Note that Definition 4.1 defines a rank join instance of two inputs and one scoring attribute, similar to $I^{2\text{–}rel} \cap I^{1\text{–}dim}$ in [29], extended though to any join predicate ϕ.

Let $\mathcal{A}$ denote the class of *deterministic rank join* algorithms that solve a k-Join instance with a behavior determined only by (i) the size of the input collections R and S, (ii) the objects $r \in R$ and $s \in S$ already examined by the algorithm, and (iii) the values of the aggregate function $\gamma(r, s)$ on pairs of these objects. In other words, an algorithm in $\mathcal{A}$ operates solely based on the knowledge it has from the objects accessed so far and it does not have any prior knowledge about the objects of the input collections. The cost of applying an algorithm $A \in \mathcal{A}$ is defined in terms of the number of objects accessed from each collection, denoted by $\texttt{topkdepth}(A, R)$ and $\texttt{topkdepth}(A, S)$.

Definition 4.2 (Access Cost) Given a k-Join instance (R, S, ϕ, γ, k) and an algorithm A, the *access cost* of A equals the total number of objects accessed from R and S:

$$\texttt{acost}(A, R, S) = \texttt{topkdepth}(A, R) + \texttt{topkdepth}(A, S)$$

In [29], HRJN*/PBRJ$^*_c$ is proven to be *instance optimal* within the class of algorithms $\mathcal{A}$ with an optimality ratio of 2, for all k-Join instances of equality join predicate. This finding is straightforwardly extended for any join predicate ϕ and thus, SFP is also *instance optimal* within class $\mathcal{A}$ with an optimality ratio of 2, i.e., there exists a constant c, such that for any k-Join instance and any $A \in \mathcal{A}$:

$$\texttt{acost}(\text{SFP}, R, S) \leq 2 \cdot \texttt{acost}(A, R, S) + c$$

Finally, we establish the optimality of BLP. Schnaitter et al. [29] present a variant of the PBRJ framework where a *block* of objects, instead of a *single* object, at a time, is accessed from the input collections. In this setup, the cost of applying a deterministic rank join algorithm A is defined with respect to the total number of accessed blocks.

Definition 4.3 (Block Access Cost) Given a k-Join instance (R, S, ϕ, γ, k) and an algorithm A, the *block access cost* of applying A equals the total number of λ sized blocks accessed from R and S:

$$\texttt{bacost}(A, R, S) = \left\lceil \frac{\texttt{topkdepth}(A, R)}{\lambda} \right\rceil + \left\lceil \frac{\texttt{topkdepth}(A, S)}{\lambda} \right\rceil$$

The instance optimality analysis conducted in [29] for HRJN*/PBRJ$^*_c$, and thus, also for SFP in case of complex join attributes, can be directly extended to the bacost metric in place of acost since the block variant of SFP processes an object similar to the original method. In practice, BLP extends and optimizes the block variant of SFP minimizing the computational cost of k-Join while employing the same bound scheme to determine termination threshold T and the same pulling strategy. Thus, BLP is, similar to the block variant of SFP, instance optimal within the class of deterministic ranked join algorithms $\mathcal{A}$ for all k-Join instances of Definition 4.1.

4.3 Extensions

We finally discuss the applicability of BLP under more general k-Join definitions.

Multiple scoring attributes per input. Under this setup, the objects of each collection, e.g., $r \in R$, are accessed in decreasing order of their aggregate score bound $\overline{\gamma}(r)$ which is defined over aggregate function γ using the values of r's scoring attributes while setting the scoring attributes of collection S to their maximum value. Although BLP can be directly applied in this setup, it is no longer instance optimal as shown in [29] due to using (similar to HRJN*/PBRJ$^*_c$) the corner bounding scheme to specify termination threshold T. To address the issue of instance optimality while providing a computationally efficient top-k join method, Finger et al. [8] proposed the FRPA algorithm as a specialization of the PBRJ framework which employs the so-called FR^* bound scheme to determine a tight threshold T and the PA pulling strategy, i.e., FRPA corresponds to PBRJ$^{PA}_{FR^*}$. As BLP extends and optimizes PBRJ in an orthogonal direction it can directly employ the bound scheme and the pulling strategy of FRPA to provide both an instance optimal and an efficient method for k-Join on complex data types with multiple scoring attributes.

Multiple inputs $R_1, \ldots, R_n$. There exist two approaches in dealing with multiple inputs. Following [17], the first option is to incrementally join the collections using multiple k-Join binary operators. Without loss of generality, consider three input collections R_1, R_2 and R_3, all sorted

by their scoring attribute score; R_3 is incrementally joined with the results of the R_1, R_2 k-Join operator. In order to access the next block on this hierarchy of k-Join operators, higher level BLP either accesses the next λ objects from R_3, or issues a λ-Join query over R_1 and R_2 to a lower-level BLP component. The accessed block is then indexed and joined with the buffered blocks exactly as discussed in Section 4.1 for the binary BLP. The second approach is to treat multiple inputs in a multi-way join fashion [29]. In this case, a multi-way variant of BLP selects the next input source R_i that satisfies $\gamma(\{b^u_{R_j}|j \neq i\}, b^\ell_{R_i}) > \max_{j \neq i} \gamma(b^\ell_{R_j}, \{b^u_{R_{j'}}|j' \neq j\})$, and accesses λ objects to form a new block b_i. Then, block b_i is indexed and joined against the indices of the buffered blocks from all input collections R_j with $j \neq i$. Last, note that in both approaches, the result set C is updated and the termination condition is examined as discussed in Section 4.1.

5 Spatial Join Attributes

In this section we investigate the application of Score-First, Join-First and Block-Based Paradigms for *spatial join attributes*. In this case, att models the spatial location of an object and takes values from the two-dimensional geographic space. As an example consider the collections of spatial objects $R = \{r_1, \ldots, r_8\}$ and $S = \{s_1, \ldots, s_8\}$ of Figure 2. The object locations are shown on the left part of the figure, while the two tables on the right list the objects in each collection in descending order of their scores. Spatial joins have been extensively studied due to their wide range of applications and their potentially high cost. Several join predicates have been proposed but in the context of this study, we consider the *spatial ϵ-distance join* predicate. The result of a *spatial distance* k-Join query denoted by k-SDJoin, involves pairs of objects (r, s) such that r is spatial close to s with respect to a given distance threshold ϵ, i.e., predicate $\phi = (dist_{SD}(r, s) \leq \epsilon)$ where $dist_{SD}$ denotes the spatial distance between the spatial attribute of r and s. Note that for the rest of the paper we consider the Euclidean distance as $dist_{SD}$.

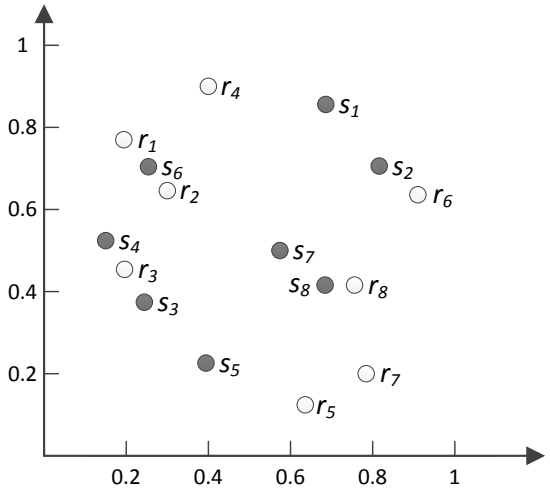
The dominant indexing structure for spatial data is the R-tree [11] which indexes minimum bounding rectangles (MBRs) of the objects hierarchically. SFP, JPF and BLP employ (in a different fashion each) an extension of the R-tree, called the aR-tree [24], which indexes both spatial locations and aggregate information of the objects. The rationale is that the aR-tree provides a way to prioritize the computation of k-SDJoin according to the aggregate scores and allows for pruning the search space based on both spatial distance predicate and aggregate score. The aR-tree has identical structure and update algorithms as the R-tree, however, each non-leaf entry is augmented with the maximum score of all objects in the subtree pointed by it. Figure 3 illustrates two aR-trees for the collections of Figure 2.

In the following, we discuss in detail how each of the three paradigms for k-Join evaluation can be realized for the k-SDJoin operator.

5.1 Applying the Score-First Paradigm

As discussed in Section 3.1, SFP incrementally accesses the objects from collection R or S in decreasing order of their scoring attributes, and joins them with the objects already examined from S or R, respectively, which for k-SDJoin are indexed by aR-trees $\mathcal{I}_S$ and $\mathcal{I}_R$. Under this perspective, each accessed object, e.g., r from R, is probed against aR-tree $\mathcal{I}_S$ to retrieve objects $s \in S$ such that pair (r, s) qualifies the join predicate $\phi = (dist_{SD}(r, s) \leq \epsilon)$ and $\gamma(r, s) > \theta$ holds, where θ equals the score of k-th candidate result pair found so far.

Procedure 1 is a pseudocode of the Probe procedure for k-SDJoin following the Score-First Paradigm. Given an object o (either $r \in R$ or $s \in S$), Probe performs an aR-tree search on the index $\mathcal{I}$ of the other collection (resp. $\mathcal{I}_S$ or $\mathcal{I}_R$) as a *score-based incremental ϵ-distance range query*. The aR-tree search is guided by a max-heap H of (o, e) pairs where e is an entry of the aR-tree $\mathcal{I}$. Max-heap H allows Probe to examine (o, e) pairs in decreasing order of their aggregate score $\gamma(o, e)$. Note that $\gamma(o, e)$ is computed using the aggregate score for entry e in the aR-tree $\mathcal{I}$ and it is an upper bound of the score of every object contained in the subtree pointed by e. During the aR-tree search, an (o, e) pair is pruned if (i) the MBR of entry e is farther than ϵ, i.e., $dist_{SD}(o, e) > \epsilon$, or (ii) $\gamma(o, e) \leq \theta$ as it would not be possible to find an object o' in the subtree pointed by e with $\gamma(o, o') > \theta$. Otherwise, pair (o, e) is inserted to the max-heap H. Finally, notice



id	att	score
r_1	(0.20, 0.78)	1.0
r_2	(0.30, 0.64)	0.8
r_3	(0.20, 0.45)	0.8
r_4	(0.40, 0.90)	0.6
r_5	(0.63, 0.12)	0.6
r_6	(0.91, 0.63)	0.4
r_7	(0.79, 0.20)	0.3
r_8	(0.76, 0.42)	0.1

Object collection R

id	att	score
s_1	(0.69, 0.85)	0.9
s_2	(0.81, 0.71)	0.9
s_3	(0.24, 0.38)	0.8
s_4	(0.15, 0.52)	0.7
s_5	(0.40, 0.22)	0.6
s_6	(0.25, 0.70)	0.4
s_7	(0.58, 0.50)	0.4
s_8	(0.68, 0.42)	0.2

Object collection S

Figure 2: Example of collections R and S with 8 spatial objects each.

that when Probe examines an (o, e) pair where e is a leaf node entry of the $\mathcal{I}$ aR-tree, i.e., e is an object (Lines 14–16), it updates the set of candidates pairs C and the θ bound which is the aggregate score of k-th candidate pair found so far.

Example 5.1 *Consider collections R and S of Figure 2 and* k-SDJoin *with $k = 1$, $\epsilon = 0.1$, and $\gamma = SUM$. First, SFP accesses r_1 from R and as aR-tree $\mathcal{I}_S$ is currently empty, r_1 is just inserted to $\mathcal{I}_R$. Then, s_1 is accessed from S and probed against $\mathcal{I}_R$ resulting in no match as $dist_{SD}(r_1, s_1) > \epsilon$. Since $\ell_R = 1.0 > \ell_S = 0.9$, r_2 is next accessed and joined (unsuccessfully) with $\mathcal{I}_S$. Similarly, s_2 and s_3 are processed, still without producing any distance join results. When r_3 is accessed and joined with $\mathcal{I}_S$ which now contains $\{s_1, s_2, s_3\}$, SFP inserts to C the first result (r_3, s_3). At this point bound $\theta = \gamma(r_3, s_3) = 1.6$ and threshold $T = \max\{\gamma(1.0, 0.8), \gamma(0.8, 0.9)\} = 1.8 > \theta$, and thus, a possibly better pair can be found and SFP cannot terminate yet. Next accessed object is r_4, which gives no join results. When s_4 is accessed and probed against aR-tree $\mathcal{I}_R$, pair (r_3, s_4) qualifies the spatial distance predicate but it is discarded as $\gamma(r_3, s_4) = 1.5 < \theta$. Then, s_5 gives no new join pairs. Finally, s_6 is retrieved without producing any join pair with an aggregate score higher than θ. However, since termination threshold $T = 1.5$, i.e., lower than θ, SFP terminates reporting $C = \{(r_3, s_3)\}$ as the final result.* ■

5.2 Applying the Join-First Paradigm

As discussed in Section 3.2, JFP primarily focuses on the join component of a k-Join query. Specifically, it first joins the input collections R and S to get the object pairs that qualify the predicate ϕ, which in case of k-SDJoin is $dist_{SD}(r, s) \leq \epsilon$, and then, retrieves the top-k pairs among them with the highest aggregate scores. For implementing the spatial distance join, we could apply algorithms either like the R-tree join [3], assuming that R and S are already indexed by R-trees, or methods

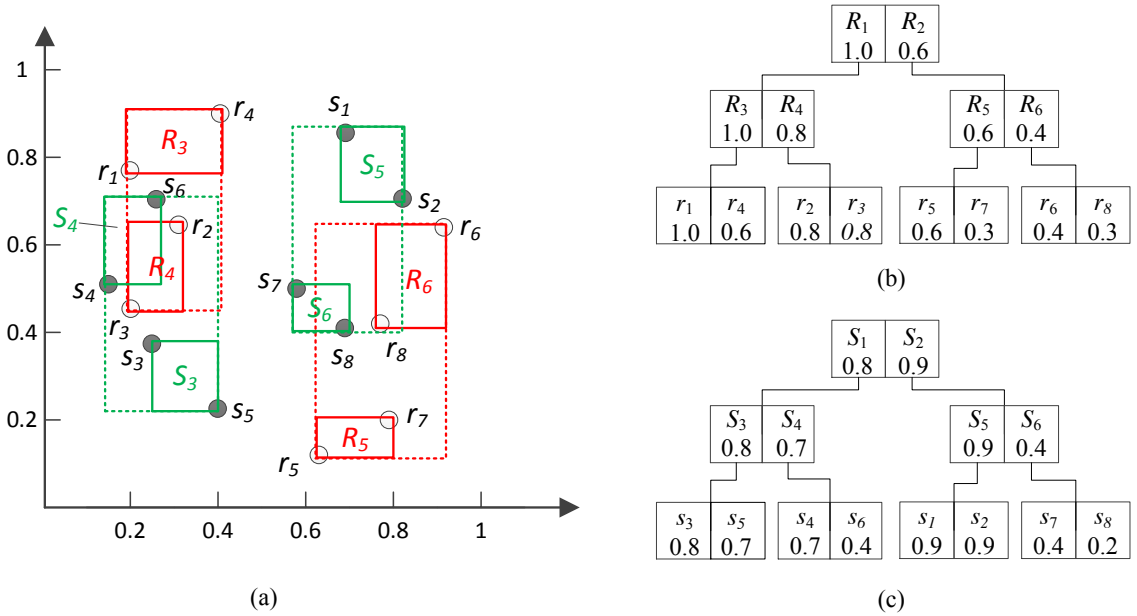


Figure 3: aR-trees for collections R and S of Figure 2.

that spatially join non-indexed inputs, like the (external memory) plane sweep algorithm [1], which first sorts R and S based on one of their coordinates and then sweeps a line along the sort axis to compute the results. However, the above approaches do not provide a way to prioritize the join result computation according to the aggregate scores of qualifying distance join pairs. Towards this direction, we present an optimized approach of applying JFP for k-SDJoin that employs aR-trees indices and manages to avoid computing the full spatial distance join between the inputs R and S.

Procedure 2 is a pseudocode of the Join procedure for k-SDJoin following the Join-First Paradigm. Given two aR-trees $\mathcal{I}_R$ and $\mathcal{I}_S$ (for input collections R and S, respectively), Join spatially joins the two trees by adapting the classic algorithm of [3] to traverse them not in a depth-first, but in a *best-first* order, which (i) still prunes entry pairs (e_R, e_S), $e_R \in \mathcal{I}_R, e_S \in \mathcal{I}_S$ for which $dist_{SD}(e_R, e_S) > \epsilon$ ($dist_{SD}$ here denotes the minimum distance between the MBRs of the two entries), but (ii) it also prioritizes the entry pairs to be examined based on $\gamma(e_R, e_S)$ (here, γ is applied on the aggregate scores stored at the entries). In other words, the entry pairs which have the maximum aggregate score are examined first during the join and this order guarantees that the qualifying object pairs will be computed incrementally in decreasing order of their aggregate scores. To achieve this, Join employs a max-heap H which initially contains all pairs of root entries within distance ϵ from each other in the two trees (Lines 3–5). Pairs of entries from H are examined (de-heaped) in priority of their aggregate scores $\gamma(e_R, e_S)$ as follows. The spatial distance join is evaluated for the corresponding aR-tree nodes and the results are inserted to H if they are non-leaf entries (branching condition at Line 10). Otherwise, if a leaf node entry pair (r, s) (i.e., object pair) is de-heaped, it is guaranteed that (r, s) has higher aggregate score than any other object pair to be found later, since entry and object pairs are accessed in decreasing order of their γ-scores from H. Therefore, the object pair is included as the next result of the k-SDJoin query to the return set C (Line 17). Finally, Join and thus JFP, terminates after k results have been computed.

Example 5.2 *Consider again the collections of Figure 2 and* k-SDJoin *with* $k = 1$, $\epsilon = 0.1$, *and* $\gamma = SUM$. *Initially, JFP creates the aR-trees of Figure 3; the entries are augmented with the maximum scores of any objects in the subtrees indexed by them (e.g.,* R_2 *has score* 0.6*). Next, JFP performs the aR-tree based spatial distance join. The roots of* $\mathcal{I}_R$ *and* $\mathcal{I}_S$ *are first considered, which adds* (R_1, S_1) *and* (R_2, S_2) *entry pairs to* H*; the other two combinations* (R_1, S_2) *and* (R_2, S_1) *are pruned by the* ϵ*-distance join predicate. The next pair to be examined is* (R_1, S_1) *because* $\gamma(R_1, S_1) = 1.8 > \gamma(R_2, S_2)$. *Thus, the nodes pointed by* R_1 *and* S_1 *are synchronously visited and their* ϵ*-distance join adds pairs* (R_3, S_4), (R_4, S_4), *and* (R_4, S_3) *to* H. *The next entry pair to be*

PROCEDURE 1: Probe

Input : o, $\mathcal{I}$, T, $\phi = (dist_{SD}(\cdot,\cdot) \leq \epsilon)$, γ, k, θ, C
Output: θ, C

```
1  initialize a max-heap H := ∅ of aR-tree entries, organized by aggregate scores;
2  for each entry e in I.root do
3      if dist_SD(o, e) ≤ ε then
4          push e into H;
5  while H ≠ ∅ and T > θ do
6      e := H.dequeue();
7      if γ(o, e) ≤ θ then
8          break;
9      if e is non-leaf node entry then
10         n := node of I pointed by e;
11         for each entry e' ∈ n do
12             if γ(o, e) > θ and dist_SD(o, e) ≤ ε then
13                 push e' into H;
14     else
15         insert (o, e) to C, remove the k-th pair in C first if |C| = k ;
16         θ := aggregate score of the k-th pair in C;
17 return ⟨θ, C⟩;
```

de-heaped is (R_3, S_4) *with* $\gamma(R_3, S_4) = 1.7$*; this results in object pair* (r_1, s_6) *being found and added to* H*. Then,* (R_4, S_3) *is de-heaped and* (r_3, s_3) *is added to* H*. The next pair to be popped from* H *is the object pair* (r_3, s_3)*; note that this is guaranteed to be the* ϵ*-distance join pair with the highest aggregate score, since it is the first object pair to be extracted from the max-heap* H*, and thus, the algorithm terminates.* ■

5.3 Applying the Block-based Paradigm

Recall from Section 4 that BLP, similar to SFP, examines the objects of the collections in decreasing order of their scoring attribute, but considers a block of objects at a time, instead of a single object. The currently accessed block is spatially joined against the blocks examined so far from the other collection, using the score bounds retained for each block to prune block pairs which may not contain k-SDJoin results. BLP joins two blocks similar to JFP. Procedure 3 illustrates the Join procedure of BLP for k-SDJoin. Notice that, different from Procedure 2 and JFP, (i) Join for BLP employs the termination threshold T in Line 5, and (ii) when the procedure identifies object pair (r, s) that qualifies the spatial ϵ-distance predicate with an aggregate score higher than bound θ, the pair is treated as a candidate result similar to SFP and thus, min-heap C and bound θ are updated accordingly in Lines 16–17.

Example 5.3 *Consider again the collections of Figure 2 and* k-SDJoin *with* $k = 1$*,* $\epsilon = 0.1$*,* $\gamma = SUM$*. For the sake of this example, the inputs are divided into blocks of 2 objects in Figure 4. First,* b_{R_1} *is read and aR-tree* $\mathcal{I}_{R_1}$ *is created. Then,* b_{S_1} *and* b_{S_2} *are accessed, bulk-loaded to* $\mathcal{I}_{S_1}$ *and* $\mathcal{I}_{S_2}$*, respectively, and joined with* $\mathcal{I}_{R_1}$ *producing no spatial join results. Next,* b_{R_2} *is read and joined with* b_{S_1} *and* b_{S_2} *(in this order), to generate* $C = \{(r_3, s_3)\}$ *and set* $\theta = 1.6$*. The next block* b_{S_3} *is joined with* b_{R_1}*, but not* b_{R_2}*, because* $\gamma(b^u_{R_2}, b^u_{S_3}) = \gamma(0.8, 0.7) = 1.5 < \theta$*; i.e., in the best case a spatial distance join between* b_{R_2} *and* b_{S_3} *will produce a pair of aggregate score* 1.5*, which is not higher than the score of current top pair* (r_3, s_3)*. Next, the join between* b_{S_3} *and* b_{R_1} *does not improve current* k-SDJoin *result. At this stage, BLP terminates because* $T = 1.5$ *is not higher than* $\theta = 1.6$*, and* $C = \{(r_3, s_3)\}$ *is returned as the final result.* ■

PROCEDURE 2: Join

Input : $\mathcal{I}_R$, $\mathcal{I}_S$, $\phi = (dist_{SD}(\cdot,\cdot) \leq \epsilon)$, γ, k
Output: C

```
1  initialize θ := −∞;
2  initialize a max-heap H of aR-tree entry pairs (e_R, e_S) organized by γ(e_R, e_S);
3  for each pair (e_R, e_S) in I_R.root × I_S.root do
4      if dist_SD(e_R, e_S) ≤ ε then
5          push (e_R, e_S) into H;
6  while H ≠ ∅ do
7      (e_R, e_S) := H.dequeue();
8      if γ(e_R, e_S) ≤ θ then
9          break;
10     if e_R and e_S are non-leaf node entries then
11         n_R := node of I_R pointed by e_R;
12         n_S := node of I_S pointed by e_S;
13         for each entry e'_R ∈ n_R and each entry e'_S ∈ n_S do
14             if γ(e_R, e_S) > θ and dist_SD(e'_R, e'_S) < ε then
15                 push (e'_R, e'_S) into H;
16     else
17         insert (r, s) to C as next k-SDJoin result;
18 return C;
```

block	id	att	score
b_{R_1}	r_1	(0.20, 0.78)	1.0
	r_2	(0.30, 0.64)	0.8
b_{R_2}	r_3	(0.20, 0.45)	0.8
	r_4	(0.40, 0.90)	0.6
b_{R_3}	r_5	(0.63, 0.12)	0.6
	r_6	(0.91, 0.63)	0.4
b_{R_4}	r_7	(0.79, 0.20)	0.3
	r_8	(0.76, 0.42)	0.1

(a) collection R

block	id	att	score
b_{S_1}	s_1	(0.69, 0.85)	0.9
	s_2	(0.81, 0.71)	0.9
b_{S_2}	s_3	(0.24, 0.38)	0.8
	s_4	(0.15, 0.52)	0.7
b_{S_3}	s_5	(0.40, 0.22)	0.7
	s_6	(0.25, 0.70)	0.4
b_{S_4}	s_7	(0.58, 0.50)	0.4
	s_8	(0.68, 0.42)	0.2

(b) collection S

Figure 4: Example of BLP with $\lambda = 2$ for k-SDJoin.

6 String Join Attributes

In this section we investigate the application of the Score-First, Join-First, and Block-Based Paradigms for *string join attributes*. Specifically, every object has a string attribute `att` that models its textual description or other types of sequential information. For example consider the collections of string objects $R = \{r_1, \ldots, r_8\}$ and $S = \{s_1, \ldots, s_8\}$ in Figure 5 which contain restaurant ratings from two different sources. Similar to spatial joins, string joins have been extensively studied due to their wide range of applications, e.g., data de-duplication and integration, collaborative filtering, entity reconciliation, and their high computational cost. Several join predicates have been proposed but in this study, we consider the *string ϵ-similarity join* predicate. A *string similarity* k-Join query denoted by k-SSJoin returns the k pairs of objects with the highest aggregate score among all pairs $(r, s) \in R \times S$ where the string attributes of r and s are similar with respect to a given distance threshold ϵ (i.e., predicate $\phi = (dist_{SS}(r, s) \leq \epsilon)$ and $dist_{SS}$ measures the distance between the string attribute of r and s). Thus, k-SSJoin focuses on identifying the most important object pairs based on their aggregate score, instead of computing the (possibly overwhelming) set of all string-similar pairs similar to conventional similarity join.

PROCEDURE 3: Join

Input : $\mathcal{I}_R$, $\mathcal{I}_S$, T, $\phi = (dist_{SD}(\cdot,\cdot) \leq \epsilon)$, γ, k, θ, C
Output: θ, C

initialize a max-heap H of aR-tree entry pairs (e_R, e_S) organized by $\gamma(e_R, e_S)$;
for *each pair* (e_R, e_S) *in* $\mathcal{I}_R.root \times \mathcal{I}_S.root$ **do**
    if $dist_{SD}(e_R, e_S) \leq \epsilon$ **then**
        push (e_R, e_S) into H;
while $H \neq \emptyset$ ***and*** $T > \theta$ **do**
    $(e_R, e_S) := H$.dequeue();
    if $\gamma(e_R, e_S) \leq \theta$ **then**
        break
    if e_R *and* e_S *are non-leaf node entries* **then**
        $n_R :=$ node of $\mathcal{I}_R$ pointed by e_R;
        $n_S :=$ node of $\mathcal{I}_S$ pointed by e_S;
        for *each entry* $e'_R \in n_R$ *and each entry* $e'_S \in n_S$ **do**
            if $\gamma(e_R, e_S) > \theta$ ***and*** $dist_{SD}(e'_R, e'_S) < \epsilon$ **then**
                push (e'_R, e'_S) into H;
    else
        insert (r, s) to C, remove the k-th pair in C first if $|C| = k$;
        $\theta :=$ aggregate score of the k-th pair in C;
return $\langle \theta, C \rangle$;

id	att	score
r_1	"*extreme_burgers*"	1.0
r_2	"*x−treme_burgers*"	0.8
r_3	"*burgermeister*"	0.8
r_4	"*dragon_snacks*"	0.6
r_5	"*the_cafe_drive*"	0.6
r_6	"*lougi's_pizza*"	0.4
r_7	"*golden_snacks*"	0.3
r_8	"*the_cake_place*"	0.1

(a) Object collection R

id	att	score
s_1	"*gourmet_food*"	0.9
s_2	"*luigi's_pizza*"	0.9
s_3	"*burgermaster*"	0.8
s_4	"*burger_meister*"	0.7
s_5	"*columbus_food*"	0.7
s_6	"*extreme_burgers*"	0.4
s_7	"*new_york_pancakes*"	0.4
s_8	"*the_cake_palace*"	0.2

(b) Object collection S

Figure 5: Example of collections R and S with 8 string objects each.

For the rest of the paper, we consider the edit distance as $dist_{SS}$; two objects r and s are similar if the string attribute of r can be transformed to the string attribute of s by at most ϵ edit operations. The set of edit operations includes replacing an element in r, deleting an element from r, and inserting an element into r. For example, under an information integration scenario k-SSJoin combines the ratings from the collections of Figure 5 to identify the k restaurants with the highest average score. To this purpose, the query needs to match the textual descriptions of a restaurant contained in both collections allowing however for some small differences (e.g., due to misspelling).

Section 6.1 details the state-of-the-art method for string ϵ-similarity joins. Then, Sections 6.2–6.4 present how the three k-Join paradigms are applied for k-SSJoin.

6.1 Background on String Similarity Joins: Pass-Join

Pass-Join [18] adopts a filter-and-refinement framework similar to the majority of the string similarity join methods in the literature. In the filtering step, the algorithm follows a partition-based approach. Particularly, given two collections of string objects R and S and an edit distance threshold ϵ, Pass-Join splits each object r in R into $\epsilon + 1$ disjoint segments based on the property that

in order for a string object s in collection S to be similar to r w.r.t. threshold ϵ, object s must contain a substring which matches a segment of r; otherwise, candidate pair (r, s) can be safely pruned.

ALGORITHM 1: Pass-Join

Input : R, S, ϵ
Output: $C = \{(r \in R, s \in R) : dist_{SS}(r, s) \leq \epsilon)\}$

```
1  sort R and S first by string length and second in alphabetical order;
2  for each r ∈ R do
3      partition r into ε + 1 segments;
4      add segments into index I_R;
5  for each s ∈ S do
6      for each inverted list L_R^{l,i} in I_R with |s| − ε ≤ l ≤ |s| + ε and 1 ≤ i ≤ ε + 1 do
7          W := SelectSubstrings(s, L_R^{l,i});
8          for w ∈ W do
9              if w is in L_R^{l,i} then
10                 C := Verify(L_R^{l,i}[w], s, ε, C) ;      // L_R^{l,i}[w] is the entry for segment w in L_R^{l,i}
11 return C;
```

Algorithm 1 illustrates the pseudocode of Pass-Join. The algorithm takes as input two collections of string objects R and S, and an edit distance threshold ϵ. The string objects in the collections are sorted first by their length and second lexicographically (Line 1). Next, an inverted index $\mathcal{I}_R$ is built on top of collection R (Lines 2–4) by dividing every object $r \in R$ into $\epsilon + 1$ segments. The inverted lists of $\mathcal{I}_R$ associate every distinct string segment with the objects that contain it. Particularly, inverted list $L_R^{l,i}$ in $\mathcal{I}_R$ indexes the i-th segment of every object with length l. Figure 6 shows inverted index $\mathcal{I}_R$ for collection R in Figure 5(a) and an edit distance threshold $\epsilon = 3$ (ignore for now the score associated to each list entry). Consider for example object r_1.att = "*extreme_burgers*" of length 15. The object is partitioned into 4 segments {"*ext*", "*reme*", "*_bur*", "*gers*"} which are added to lists $L_R^{15,1}$, $L_R^{15,2}$, $L_R^{15,3}$, $L_R^{15,4}$, respectively. If more than one objects contain a segment w, a *bucket* entry is added to the corresponding inverted list. Observe bucket entries $\langle$"*_bur*", $\{r_1, r_2\}\rangle$ and $\langle$"*gers*", $\{r_1, r_2\}\rangle$ of lists $L_R^{15,3}$ and $L_R^{15,4}$, respectively in Figure 6, for objects r_1.att = "*extreme_burgers*" and r_2.att = "*x−treme_burgers*", which both contain segments "*_bur*" and "*gers*".

After indexing collection R, Pass-Join iterates over the objects in collection S and computes the join result C by probing the $\mathcal{I}_R$ index (Lines 5–10). According to the *length filtering* introduced in [10], every object s in S can be only joined with objects in R of length l, such that $l \geq |s| - \epsilon$ and $l \leq |s| + \epsilon$, where $|s|$ denotes the length of the string object s. To access such objects in R, Pass-Join traverses every inverted list $L_R^{l,i}$ with $|s| - \epsilon \leq l \leq |s| + \epsilon$ and $1 \leq i \leq \epsilon + 1$ (Line 6). The contents of every $L_R^{l,i}$ list are filtered keeping only the objects whose segments can match at least one of the substrings of s in set W (Lines 8–9). The substrings of s contained in set W are selected in a *multi-match-aware* manner employed by the SelectSubstrings procedure (Line 7). Finally, Pass-Join verifies candidate object pairs (r, s) for every object r in $L_R^{l,i}$ that contains a matched segment/substring w, using a length-based and an extension-based verification method (Line 10).

In the following, we adapt Pass-Join to compute k-SSJoin using the Score-First, Join-First and Block-based Paradigms. For this purpose and similar to the aggregate score information on the aR-tree, we extend the $\mathcal{I}_R$ index of Pass-Join including inside each list (bucket) entry the *maximum score* of the involved objects. Consider inverted index $\mathcal{I}_R$ in Figure 6. Bucket entry $\langle$"*the*", $\{r_5, r_8\}\rangle$ in list $L_R^{14,1}$ is assigned a score of 0.6 which equals r_5.score, since r_5.score > r_8.score.

inverted list $L^{l,i}$	entries
$L^{13,1}$	$\langle$"*bur*", $\{r_3\}, 0.8\rangle\langle$"*dra*", $\{r_4\}, 0.6\rangle\langle$"*lou*", $\{r_6\}, 0.4\rangle$ $\langle$"*gol*", $\{r_7\}, 0.3\rangle$
$L^{13,2}$	$\langle$"*ger*", $\{r_3\}, 0.8\rangle\langle$"*gon*", $\{r_4\}, 0.6\rangle\langle$"*gi*", $\{r_6\}, 0.4\rangle$ $\langle$"*den*", $\{r_7\}, 0.3\rangle$
$L^{13,3}$	$\langle$"*mei*", $\{r_3\}, 0.8\rangle\langle$"*_sn*", $\{r_4$,$r_7\}, 0.6\rangle\langle$"*s_p*", $\{r_6\}, 0.4\rangle$
$L^{13,4}$	$\langle$"*ster*", $\{r_3\}, 0.8\rangle\langle$"*acks*", $\{r_4, r_7\}, 0.6\rangle$ $\langle$"*izza*", $\{r_6\}, 0.4\rangle$
$L^{14,1}$	$\langle$"*the*", $\{r_5, r_8\}, 0.6\rangle$
$L^{14,2}$	$\langle$"*_ca*", $\{r_5, r_8\}, 0.6\rangle$
$L^{14,3}$	$\langle$"*fe_d*", $\{r_5\}, 0.6\rangle, \langle$"*ke_p*", $\{r_8\}, 0.1\rangle$
$L^{14,4}$	$\langle$"*rive*", $\{r_5\}, 0.6\rangle, \langle$"*lace*", $\{r_8\}, 0.1\rangle$
$L^{15,1}$	$\langle$"*ext*", $\{r_1\}, 1.0\rangle, \langle$"*x−t*", $\{r_2\}, 0.8\rangle$
$L^{15,2}$	$\langle$"*reme*", $\{r_1, r_2\}, 1.0\rangle$
$L^{15,3}$	$\langle$"*_bur*", $\{r_1, r_2\}, 1.0\rangle$
$L^{15,4}$	$\langle$"*gers*", $\{r_1, r_2\}, 1.0\rangle$

Figure 6: Inverted index $\mathcal{I}_R$ on collection R in Figure 5 and an edit distance threshold $\epsilon = 3$.

6.2 Applying the Score-First Paradigm

As discussed in Section 3.1, SFP incrementally accesses the objects of the input collection R or S in decreasing order of their scoring attribute, and joins them with the objects already examined from S or R. Under this perspective, to employ Pass-Join in the context of SFP, we need to make two adjustments: (i) the objects in each collection are no longer sorted first by their lengths and second lexicographically but according to their scoring attribute, and (ii) instead of indexing *only* collection R building *offline* $\mathcal{I}_R$, *two* inverted indices $\mathcal{I}_R$ and $\mathcal{I}_S$ are *incrementally* built *online* to buffer the objects examined from collections R and S, respectively. Hence, the currently accessed object, e.g., r from R, is first probed against the $\mathcal{I}_S$ inverted index to retrieve objects $s \in S$ such that pair (r, s) qualifies the join predicate $\phi = (dist_{SS}(r, s) \leq \epsilon)$ and $\gamma(r, s) > \theta$ holds, where θ equals the score of k-th candidate result pair found so far, and then, r is divided into $\epsilon+1$ segments and indexed by $\mathcal{I}_R$ according to the rationale of Pass-Join.

Procedure 4 illustrates a pseudocode of the Probe procedure for k-SSJoin. The functionality of Probe is reminiscent to the probing part of Algorithm 7 in Lines 6–10. The current object $o = r$ (assume without loss of generality that o is from R) is joined with already examined objects s in S of length l that is larger than or equal to $|r| - \epsilon$ and smaller than or equal to $|r| + \epsilon$, provided that the segments of s can match at least one of the substrings of r. Different from Pass-Join, though, the objects s from S that contain a match substring w of r, i.e., the contents of the $L_S^{l,i}[w]$ entry, are further filtered using the $\gamma(o.\texttt{score}, L^{l,i}[w].\texttt{score}) > \theta$ condition, where $L^{l,i}[w].\texttt{score}$ equals the highest score of the objects in $L^{l,i}[w]$ (Line 5). Finally, Pass-Join employs the Verify procedure to verify the candidate object pairs similar to Pass-Join. Verify for SFP also updates set C and the θ bound which equals the aggregate score of k-th candidate pair found so far.

Example 6.1 *Consider collections R and S of Figure 5 and* k-SSJoin *with $k = 1$, $\epsilon = 3$, and $\gamma = SUM$. SFP first accesses r_1 from R. As index $\mathcal{I}_S$ is empty, r_1 is split in $\epsilon + 1 = 4$ segments which are inserted to $\mathcal{I}_R$: $L_R^{15,1} = \{\langle$"ext", $\{r_1\}, 1.0\rangle\}$, $L_R^{15,2} = \{\langle$"reme", $\{r_1\}, 1.0\rangle\}$, $L_R^{15,3} = \{\langle$"_bur", $\{r_1\}, 1.0\rangle\}$, $L_R^{15,4} = \{\langle$"gers", $\{r_1\}, 1.0\rangle\}$. Next, s_1 is accessed from S and probed against $\mathcal{I}_R$ without producing any join results as no substring of s_1 matches the existing segments in $\mathcal{I}_R$. Since $\ell_R = 1.0 > \ell_S = 0.9$, r_2 is the next object to be accessed and joined (unsuccessfully) with $\mathcal{I}_S$. Similarly, s_2 and s_3 are accessed in turn, still without producing any string join results. When r_3 is accessed and probed against $\mathcal{I}_S$ (now containing entries for s_1, s_2 and s_3), the substring "bur" of r_3 is matched with entry $L_S^{12,1}$["bur"] that contains s_3. Verification confirms that $dist_{SS}(r_3, s_3) = 2 < \epsilon$, and as bound θ is not yet defined, SFP inserts to C the first join result (r_3, s_3) and sets $\theta = \gamma(r_3, s_3) = 1.6$. Currently, $T = \max\{\gamma(1.0, 0.8), \gamma(0.8, 0.9)\} = 1.8 > \theta$, which means that a possibly better object pair can be found and SFP cannot terminate yet. The next accessed object*

PROCEDURE 4: Probe

Input : o, $\mathcal{I}$, T, $\phi = (dist_{SS}(\cdot,\cdot) \leq \epsilon)$, γ, k, θ, C
Output: θ, C

```
1 for each inverted list L^{l,i} in I with |o| − ε ≤ l ≤ |o| + ε and 1 ≤ i ≤ ε + 1 and while T > θ
  do
2     W := SelectSubstrings(o, L^{l,i});
3     for w ∈ W do
4         if w is in L^{l,i} then
5             if γ(o.score, L^{l,i}[w].score) > θ then
6                 ⟨θ, C⟩ := Verify(L^{l,i}[w], o, ε, θ, C);
7 return ⟨θ, C⟩;
```

is r_4, which gives no new join results. Then, s_4 is accessed and probed against $\mathcal{I}_R$. Note that although a substring of s_4 matches a segment of r_3 and in fact $dist_{SS}(r_3, s_4) = 1 < \epsilon$, the pair (r_3, s_4) cannot be part of the result as $\gamma(r_3, s_4) = 1.5 < \theta$. Next, s_5 gives no new join pairs. Finally, s_6 is retrieved without producing a join pair with aggregate score higher than θ, similar to the case of s_4. However, since termination threshold T is now set to 1.5, i.e., lower than θ, SFP terminates reporting $C = \{(r_3, s_3)\}$ as the final result. ■

6.3 Applying the Join-First Paradigm

To compute k-SSJoin following the rationale of JFP, we directly employ the Pass-Join algorithm discussed in Section 6.1 to get the object pairs that qualify the predicate $\phi = (dist_{SS}(r, s) \leq \epsilon)$. In Section 3.2, JFP is described as an index join for the sake of generality, but for string join attributes we index only R and probe every object in S against $\mathcal{I}_R$, according to Pass-Join. For probing $\mathcal{I}_R$, we apply the Probe procedure introduced in the previous section for SFP by setting termination threshold $T = \infty$. Finally, we use bounds to avoid computing the entire string join between the input collections, accelerating the computation of k-SSJoin. The idea is to sort the objects of collection S in decreasing order of their `score` attribute; when JFP accesses an object s it checks whether s can contribute to a join result of aggregate score higher than the θ threshold, i.e., the score of the k-th object pair found so far. If not, JFP terminates, because all objects from S not examined yet have score equal to or lower than current object s. Procedure 5 is a pseudocode of the Join procedure for k-SSJoin following the Join-First Paradigm. For the termination condition in Line 4, we consider the object r_{max} with the highest score in R, and therefore, $\gamma(r_{max}, s)$ is an upper bound of the aggregate score a join pair that includes current object s could have.

PROCEDURE 5: Join

Input : $\mathcal{I}_R$, S, $\phi = (dist_{SS}(\cdot,\cdot) \leq \epsilon)$, k
Output: C

```
1 initialize θ := −∞;
2 sort S in descending order of the score attribute;
3 for each s in S do
4     if γ(r_max, s) ≤ θ then
5         break;   // r_max is the object in R with the highest scoring attribute.
6     ⟨θ, C⟩ := Probe(s, I_R, ∞, (dist_SS(·,·) ≤ ε), γ, k, θ, C);
7 return C;
```

Example 6.2 *Consider again collections R and S of Figure 5 and* k-SSJoin *with $k = 1$, $\epsilon = 3$, and $\gamma = SUM$. As a first step, JFP partitions the string attribute of the objects in R into $\epsilon+1 = 4$ segments and builds the $\mathcal{I}_R$ inverted index of Figure 6. It also sorts the objects of S in descending order of their scores. Next, JFP performs the string join by probing first object s_1 against $\mathcal{I}_R$ which*

produces no join result. Then, s_2 is successfully joined with r_6 as $dist_{SS}(r_6, s_2) = 1$; thus, the result pair (r_6, s_2) is inserted into C and $\theta = \gamma(r_6, s_2) = 1.3$. When s_3 is examined, JFP considers the best possible result pair (r_1, s_3) combined with $r_1 = r_{max}$ from collection R. Since, $\gamma(r_1, s_3) = 1.8 > \theta$, JFP cannot terminate; thus, s_3 is probed against $\mathcal{I}_R$. At this point, the candidate join pair (r_3, s_3) which has higher aggregate score than current θ is identified and hence, (r_3, s_3) replaces (r_6, s_2) in C and $\theta = \gamma(r_3, s_3) = 1.6$. In the following, objects s_4 and s_5 are considered without however producing any better results than (r_3, s_3). Finally, when s_6 is accessed and the best possible result pair (r_1, s_6) is considered, $\gamma(r_1, s_6) = 1.4 < \theta$ holds; thus, JFP terminates ignoring the rest of the objects in S and reporting $C = \{(r_3, s_3)\}$ as the final result. ■

6.4 Applying the Block-based Paradigm

BLP operates as an adaptation of both SFP and JFP at the block level. To compute k-SSJoin, the objects of input collections R and S, sorted in decreasing order of their `score` attribute similar to SFP, are in blocks of size λ. BLP accesses one block of objects at a time from either of the collections. Following the rationale of Pass-Join for string joins, the current block is indexed only if it originates from R, i.e., b_R, while in both cases the current block b_R or b_S is joined against the blocks already accessed from collection S or R, respectively.[2] Recall at this point that BLP utilizes the score bounds retained for each block to avoid computing every possible block-level join, and consequently, the entire string join (Line 11 in Paradigm 3). The process of joining two blocks b_R and b_S is similar to the one employed by JFP, i.e., every object s in b_S is probed against inverted index $\mathcal{I}_{b_R}$. Note that different from JFP, objects in b_S are already in decreasing order of their `score` attribute. Procedure 6 illustrates the pseudocode of the Join procedure for k-SSJoin following the Block-based Paradigm. Similar to JFP, Join for BLP employs a breaking condition to terminate a block-level join between b_R and b_S if for current object s in b_S, $\gamma(b_R^u, s.\texttt{score}) \leq \theta$ holds, i.e., s cannot contribute to a join result with an aggregate score higher than the score of the k-th object pair found so far; recall that b_R^u is the highest score of the objects contained in block b_R. Join for BLP uses the Probe procedure of SFP but, different to JFP, it also utilizes termination threshold T.

PROCEDURE 6: Join

Input : $\mathcal{I}_{b_R}$, b_S, T, $\phi = (dist_{SS}(\cdot,\cdot) \leq \epsilon)$, γ, k, θ, C
Output: θ, C

```
1 for each object s in block b_S do
2     if γ(b_R^u, s.score) ≤ θ then
3         break;
4     ⟨θ, C⟩ := Probe(s, I_{b_R}, T, (dist_SS(·,·) ≤ ε), γ, k, θ, C);
5 return ⟨θ, C⟩;
```

Example 6.3 *Consider again collections R and S in Figure 5 and k-SDJoin with $k = 1$, $\epsilon = 3$, $\gamma = SUM$. For the sake of this example, the collections are partitioned into blocks of 2 objects each, as pictured in Figure 7. Initially, block b_{R_1} is read and inverted index $\mathcal{I}_{b_{R_1}}$ is built by dividing the string attribute of the contained objects into $\epsilon + 1 = 4$ segments; we have $L^{15,1}_{b_{R_1}} = \{\langle$"ext", $\{r_1\}, 1.0\rangle, \langle$"x−t", $\{r_2\}, 0.8\rangle\}$, $L^{15,2}_{b_{R_1}} = \{\langle$"reme", $\{r_1, r_2\}, 1.0\rangle\}$, $L^{15,3}_{b_{R_1}} = \{\langle$"_bur", $\{r_1, r_2\}, 1.0\rangle\}$, and $L^{15,4}_{b_{R_1}} = \{\langle$"gers", $\{r_1, r_2\}, 1.0\rangle\}$. Then, b_{S_1} is accessed and objects s_1 and s_2 are probed against $\mathcal{I}_{b_{R_1}}$ producing no join results. Similarly, b_{S_2} is accessed and joined (unsuccessfully) with b_{R_1}. After reading b_{R_2}, the block is joined with b_{S_1} and b_{S_2} (in this order). When joining with b_{S_2}, the substring "bur" in s_3 is matched with the $L^{13,1}_{b_{R_2}}$["bur"] entry that contains object r_3. Further verification confirms that $dist_{SS}(r_3, s_3) = 2 < \epsilon$ and thus, the (r_3, s_3) pair is inserted into C and $\theta = \gamma(r_3, s_3) = 1.6$. In addition, index $\mathcal{I}_{b_{R_2}}$ for current block b_{R_2} is built. The next block b_{S_3} is joined with b_{R_1} but not with b_{R_2}, because $\gamma(b^u_{R_2}, b^u_{S_3}) = \gamma(0.8, 0.7) = 1.5 < \theta = 1.6$. Specifically for*

[2] We also experimented with a version of BLP that uses two inverted indices, which however was always less efficient.

block	id	att	score
b_{R_1}	r_1	"extreme_burgers"	1.0
	r_2	"x-treme_burgers"	0.8
b_{R_2}	r_3	"burgermeister"	0.8
	r_4	"dragon_snacks"	0.6
b_{R_3}	r_5	"the_cafe_drive"	0.6
	r_6	"lougi's_pizza"	0.4
b_{R_4}	r_7	"golden_snacks"	0.3
	r_8	"the_cake_place"	0.1

(a) collection R

block	id	att	score
b_{S_1}	s_1	"gourmet_food"	0.9
	s_2	"luigi's_pizza"	0.9
b_{S_2}	s_3	"burgermaster"	0.8
	s_4	"burger_meister"	0.7
b_{S_3}	s_5	"columbus_food"	0.7
	s_6	"extreme_burgers"	0.4
b_{S_4}	s_7	"new_york_pancakes"	0.4
	s_8	"the_cake_palace"	0.2

(b) collection S

Figure 7: Example of BLP with $\lambda = 2$ for k-SSJoin.

b_{S_3}, probing s_5 against $\mathcal{I}_{b_{R_1}}$ does not improve the current k-SSJoin *result while s_6 is not probed against $\mathcal{I}_{b_{R_1}}$ at all because $\gamma(b^u_{R_1}, s_6.score) = \gamma(r_1, s_6) = 1.4 < \theta$, i.e., s_6 is not able to produce join results of aggregate score higher than θ. At this stage, BLP terminates as $T = 1.5$ is not higher than $\theta = 1.6$, and $C = \{(r_3, s_3)\}$ is returned as the final result.* ■

7 Models

An issue that remains open is how to determine an appropriate block size λ for BLP. We model this selection as an optimization problem.

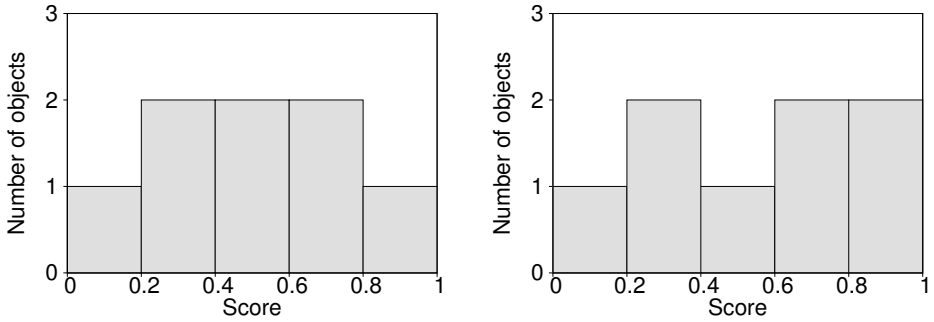
7.1 Selecting Block Size λ

The optimal value of λ minimizes the cost of computing a k-Join query with BLP, captured by the *objective cost function* $\mathcal{C}(\lambda)$:

$$\mathcal{C}(\lambda) = |N_{index}(\lambda)| \cdot \mathcal{C}_{index}(\lambda) + |N_{join}(\lambda)| \cdot \mathcal{C}_{join}(\lambda) \tag{1}$$

Intuitively, the $|N_{index}(\lambda)| \cdot \mathcal{C}_{index}(\lambda)$ part of the objective function equals the total indexing cost for BLP with $|N_{index}(\lambda)|$ being the total number of blocks indexed from input collections R and S, and $\mathcal{C}_{index}(\lambda)$ being the cost of indexing each of these blocks. On the other hand, the $|N_{join}(\lambda)| \cdot \mathcal{C}_{join}(\lambda)$ part equals the total joining cost for BLP with $|N_{join}(\lambda)|$ being the total number of block-level joins performed, and $\mathcal{C}_{join}(\lambda)$ being the cost of joining two blocks. In the following, we first elaborate on each factor of the $\mathcal{C}(\lambda)$ objective function and then we discuss how the value of λ that minimizes $\mathcal{C}(\lambda)$ can be estimated.

Regarding $\mathcal{C}_{index}(\lambda)$ and $\mathcal{C}_{join}(\lambda)$, both costs are determined by the join attribute type and the methodology to perform a block-level join. In Section 8, we conduct an experimental analysis of BLP for the case studies of spatial and string join attributes, and we discuss in detail how $\mathcal{C}_{index}(\lambda)$ and $\mathcal{C}_{join}(\lambda)$ can be empirically estimated. Next, we consider the $|N_{index}(\lambda)|$ factor. Following the analysis in Section 4.2, BLP terminates after accessing blocks $b_{R_{\lceil d_R/\lambda \rceil}}$ and $b_{S_{\lceil d_S/\lambda \rceil}}$



(a) $\mathcal{H}_R$ (b) $\mathcal{H}_S$

Figure 8: Example of score histograms for the object collections in Figure 2.

which contain objects r_{d_R} and s_{d_S}, respectively; r_{d_R} and s_{d_S} are the last objects examined by SFP with $d_R = \texttt{topkdepth}(\text{SFP}, R)$ and $d_S = \texttt{topkdepth}(\text{SFP}, S)$. Thus, BLP will index

$$|N_{index}(\lambda)| = \lceil d_R/\lambda \rceil + \lceil d_S/\lambda \rceil \tag{2}$$

blocks, in total. Finally, we consider the $|N_{join}(\lambda)|$ factor. To determine the total number of block-level joins performed by BLP we employ the notion of *any-k depth* c_R and c_S introduced in [13] as the minimum number of objects examined from each input collection in order to identify the first k candidate object pairs. Note that the first k candidate object pairs are not necessarily among the final k-Join results, and that by definition $c_R \leq d_R$ and $c_S \leq d_S$. With c_R and c_S, the execution of BLP can be divided in two parts:

(i) Until the $b_{R_{\lceil c_R/\lambda \rceil}}$ and $b_{S_{\lceil c_S/\lambda \rceil}}$ blocks containing objects r_{c_R} and s_{c_S}, respectively, are examined, BLP cannot employ the $\gamma(b^u_{R_i}, b^u_{S_j}) \leq \theta$ pruning mechanism as there are less than k candidate object pairs identified so far and the θ threshold that equals the aggregate score of the current k-th best candidate object pair is not yet defined. Thus, every possible block-level join $b_{R_i} \bowtie_\phi b_{S_j}$ is to be computed, resulting in a total number of $\lceil c_R/\lambda \rceil \cdot \lceil c_S/\lambda \rceil$ block-level joins.

(ii) After blocks $b_{R_{\lceil c_R/\lambda \rceil}}$ and $b_{S_{\lceil c_S/\lambda \rceil}}$ are examined, threshold θ can be defined based on the score of the r_{c_R} and s_{c_S} objects, i.e., $\theta = \gamma(r_{c_R}, s_{c_S})$ and consequently, the $\gamma(b^u_{R_i}, b^u_{S_j}) \leq \theta$ pruning criterion for block-level joins can be applied. Thus, to approximate the total number of block-level joins to be performed by BLP after r_{c_R} and s_{c_S}, denoted by $|N^{c_R,c_S}_{join}(\lambda)|$, we need to estimate first the score of the r_{c_R} and s_{c_S} objects for computing threshold θ, and second, the upper score bound of every block, i.e., the score of the very first object in the block, for determining which block-level joins qualify the pruning criterion. For this purpose, we employ histogram statistics from the input collections R and S. In particular, we assume that equi-width histograms $\mathcal{H}_R$ and $\mathcal{H}_S$ summarize the *score* distributions in R and S. Unlike the expensive multi-dimensional histograms employed by [6, 30], 1-dimensional $\mathcal{H}_R$ and $\mathcal{H}_S$ can be derived in low cost from the initial inputs based on an independence assumption or via sampling. Note that in case collections R and S are not pre-sorted on their scoring attribute (as required by BLP), $\mathcal{H}_R$ and $\mathcal{H}_S$ can be also computed during the sorting process. Figure 8 illustrates exemplary $\mathcal{H}_R$ and $\mathcal{H}_S$ for the collections in Figure 2. A single pass over $\mathcal{H}_R$ suffices to identify which interval contains the score of an object r_i, e.g., r_{c_R} or the first inside a b_R block. Since the objects in R are sorted in decreasing order of their `score` attribute, r_i is the i-th object of the collection. Without loss of generality, we finally set the score of r_i equal to the upper bound of the score interval. Consider for instance $\mathcal{H}_R$ in Figure 8 and assume that we want to estimate the upper score bound of the b_{R_2} block in Figure 4. By definition, $b^u_{R_2}$ equals the score of the first object in b_{R_2}, i.e., r_3, which is the 3rd object of collection R. As the $(0.8, 1.0]$ and the $(0, 6, 0.8]$ score intervals of $\mathcal{H}_R$ contain 1 and 2 objects, respectively, the score of r_3 should fall inside $(0, 6, 0.8]$ which gives an estimation of 0.8. Note that the actual score of r_3 is 0.8.

Combining the two parts, BLP will perform

$$|N_{join}(\lambda)| = \lceil c_R/\lambda \rceil \cdot \lceil c_S/\lambda \rceil + |N_{join}^{c_R,c_S}(\lambda)| \quad (3)$$

block-level joins, in total.

With objective cost function $\mathcal{C}(\lambda)$ defined using Equations (1)–(3), we now discuss how the λ value that minimizes $\mathcal{C}(\lambda)$ can be estimated. Following the definition of top-k depth in Section 4.2, we do not need to consider more than $max\{d_R, d_S\}$ objects inside each block, i.e., $\lambda \in [1, max\{d_R, d_S\}]$, since no more that d_R and d_S objects from collections R and S, respectively, are examined to compute the k-Join results. In fact with $\lambda = 1$, BLP operates exactly as SFP, while with $\lambda = max\{d_R, d_S\}$, BLP operates as an improved JFP that joins only the first (in decreasing order of their score) d_R and d_S objects of the collections. To efficiently determine a good value for λ inside $[1, max\{d_R, d_S\}]$, we employ the *golden section search technique* proposed in [15]. The idea behind this technique is to progressively narrow the range where optimal λ value lies inside following a divide and conquer approach, until a good estimation of this value is found.

7.2 Estimating any-k and top-k Depths

Finally, we discuss how depths c_R, d_R, c_S and d_S are estimated. One option is to adopt the model proposed in [13] for this purpose (recall that in the absence of multi-dimensional statistics, the [30] model operates similar to [13]). Specifically, assuming that $c_R \cdot c_S \cdot \sigma \approx k$, any-$k$ depths are set to $c_R = c_S = \sqrt{k/\sigma}$, where σ is the join selectivity of the input collections, and top-k depths are set to $d_R = d_S = 2 \cdot c_R = 2 \cdot \sqrt{k/\sigma}$. Join selectivity σ is computed via sampling inputs R and S. This model however is of limited applicability; it is based on the assumptions that the values of both the join and the scoring attributes should follow a uniform distribution and that there exists no correlation between these two attributes. On one hand, with a uniformly distributed join attribute a good estimation for join selectivity σ is achieved via sampling and thus, also a good estimation of any-k depths c_R and c_S. On the other hand, a uniformly distributed scoring attribute is required for estimating d_R and d_S as twice the any-k depth. In practice, however, either of these assumptions may not hold. For instance, as pointed out even in [13], in a hierarchy of joins where the output of one k-Join operator serves as input to another, the score distributions of higher level joins tend to be normal. Although formulas for computing c_R, c_S, d_R and d_S for the high level joins are provided, the work in [13] still requires knowledge of the join hierarchy and the distribution for both join and scoring attributes, which limits the applicability of the proposed model. Further, our analysis in Section 8 on real-world collections shows that a correlation between the join and the scoring attribute may exist. As an example, consider the collections of spatial objects in Figure 1(a). Hotels located in the city center close to important landmarks are usually assigned higher scores compared to the rest of the hotels.

Under this, we devise a novel approach for estimating any-k and top-k depths which uses cheap-to-compute statistics and is able to better cope with the special characteristics of the inputs. In Section 8, we experimentally show the superiority of our model over the work in [13]. We first discuss any-k depths. Our analysis on real data showed that c_R, c_S are determined by the join selectivity of the upper part of the sorted inputs, i.e., the highly scored objects, and is usually different from the selectivity of the entire collections. The idea behind our approach is to repeatedly sample the upper part of each input and compute the join selectivity of these parts until the potential number of results, denoted by k', becomes $k \leq k' \leq \delta \cdot k$, where k is the number of desired results for k-Join and δ is a tuning threshold parameter to avoid the $k' \gg k$ case. Specifically, we initially focus on the first t objects in each collection and estimate their join selectivity σ_t via sampling; t is a tuning parameter but alternatively any-k depth computed by [13] can be employed in its place. Based on the potential number of results $k' = t^2 \cdot \sigma_t$ we distinguish between two cases. If $k' < k$, we need to join a larger part of the collections to identify the first k candidate object pairs, i.e., $c_R, c_S > t$, and thus, we increase t by ξ^+ times and repeat the process for the new $t = \xi^+ \cdot t$ value. Otherwise, if $k' > \delta \cdot k$ although we have enough join results we choose to decrease t by ξ^- times and repeat the process to get a better estimation for c_R, c_S; note that tuning parameters ξ^+ and ξ^- are selected such that $\xi^+ > 1 > \xi^- > 0$. The above process terminates when $k \geq k' \geq \delta \cdot k$, and current t is returned as the estimation of both c_R and c_S. Finally, to estimate top-k depths d_R and d_S, we employ the $\theta = \gamma(r_{c_R}, s_{c_S})$ threshold defined by BLP after

accessing blocks $b_{R_{\lceil c_R/\lambda \rceil}}$ and $b_{S_{\lceil c_S/\lambda \rceil}}$. Specifically, d_R equals the number of r_i objects in collection R whose aggregate score γ with the highest scored object in S, i.e., s_1, is higher than threshold θ. Depth d_S is defined in a similar manner. Formally, we have:

$$\begin{aligned} d_R &= |\{r_i \in R : \gamma(r_i, s_1) > \theta\}| \\ d_S &= |\{s_j \in S : \gamma(r_1, s_j) > \theta\}| \end{aligned} \tag{4}$$

To evaluate Equation 4 we employ again the $\mathcal{H}_R$ and $\mathcal{H}_S$ score histograms on the input collections, as follows. Without loss of generality we focus on d_R. Following Equation 4, the goal is to find the last record r_i in R with $\gamma(r_i, s_1) > \theta$. Given threshold $\theta = \gamma(r_{c_R}, s_{c_S})$ and the score of s_1 we first deduce a lower bound ℓ_R for the score of such an object r_i, and then identify the score interval of $\mathcal{H}_R$ where ℓ_R falls inside. With this interval, we also get how many intervals involve scores higher than ℓ_R and hence, how many objects in collection R have a score higher than ℓ_R. The aggregate score of these objects with s_1 is by definition higher than θ. Consider now the example in Figure 4 and the histograms in Figure 8. For simplicity, assume that we have already estimated any-k depths $c_R = c_S = 2$, and thus, threshold $\theta = \gamma(r_{c_R}, s_{c_S}) = \gamma(r_2, s_2) = SUM(0.8, 0.8) = 1.6$. Hence, in order for an object r_i in R to have an aggregate score with s_1 higher than threshold θ, i.e., $\gamma(r_i, s_1) = \gamma(r_i, 0.9) > 1.6$, its score needs to be higher than $\ell_R = 0.7$. According to $\mathcal{H}_R$, score bound ℓ_R falls inside interval $(0.6, 0.8]$. By assuming similar to the case of estimating c_R and c_S that all objects whose score falls inside $(0.6, 0.8]$ have a score equal to the upper bound 0.8, collection R contains approximately 3 objects with score higher than ℓ_R, i.e., 2 from score interval $(0.6, 0.8]$ and 1 from $(0.8, 1.0]$, and therefore, we set $d_R = 3$.

8 Experimental Evaluation

In this section, we experimentally evaluate our methodology for processing k-Join on complex data types and particularly, the cases of spatial and string join attributes. First, Section 8.1 details the setup of our evaluation. Section 8.2 justifies our decision to focus on the efficient, in terms of CPU cost, k-Join evaluation, while Section 8.3 our decision to use aR-trees as the indexing structure for k-SDJoin. Sections 8.4 and 8.5 demonstrate the effectiveness of our model on estimating any-k and top-k depths, and of our model for estimating the optimal block size λ used in BLP, respectively. Finally, Section 8.6 carries out an extensive comparison of the Score-First, Join-First and Block-based evaluation paradigms. All methods involved in this study were implemented in C++ and the experiments were conducted on a 2.3 GHz Intel Core i7 CPU with 8GB of RAM running OS X.

8.1 Experimental Setup

Our experimental analysis involves both real and synthetic collections of score-carrying objects. Specifically, regarding the real-world collections:

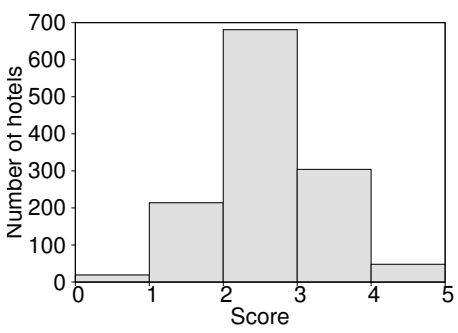
(i) For k-SDJoin, we used a collection of 645K hotels from Booking.com (BHOTELS) and a set of 481K restaurants from TripAdvisor.com (RESTS). Join attribute `att` stores the location of an object in 2-dimensional space and `score` is a user-generated rating. We denote this test by BHOTELS-RESTS.

(ii) For k-SSJoin, we used BHOTELS and a set of 255K from TripAdvisor.com (THOTELS). Attribute `score` is a user-generated rating and attribute `att` stores the name of a hotel.

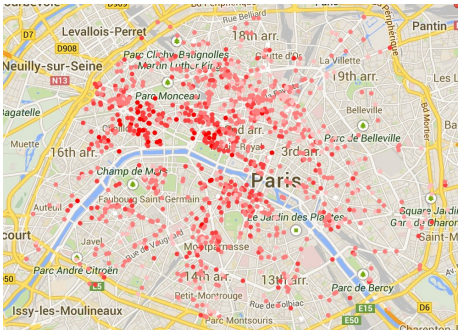
Due to the limited size of the real-world datasets, we also generated collections with synthetic scores employing both real and synthetic join attributes. Specifically:

(i) For k-SDJoin, we used the FLICKR collection of 1.68M locations associated with photographs taken from the city of London, UK over a period of 2 years and hosted on flickr.com, and the ISLES collection of 20M POIs in the area of the British Isles drawn from openstreetmap.org.

(ii) For k-SSJoin, we used the CITIES collection of 1M geographical names taken from World Gazetteer with a 200 symbols dictionary and a non-uniform distribution of the strings length



(a) Score distribution



(b) Hotels' location and score

Figure 9: Hotels in Paris with user-generated scores from TripAdvisor.com.

description	parameter	values
Join selectivity	ϵ_{SD}	$0.0001, 0.0005, \mathbf{0.001}, 0.005, 0.01$
	ϵ_{SS}	READS: $0, 4, \mathbf{8}, 12, 16$ BHOTELS-THOTELS: $0, 1, \mathbf{2}, 3, 4$ CITIES: $0, 1, \mathbf{2}, 3, 4$
Number of results	k	$1, 5, \mathbf{10}, 50, 100$
Number of seeds	$\|\Sigma\|$	CORR: $10, \mathbf{20}, 50, 100$
Number of objects ($\times$1,000,000)	$\|R\| + \|S\|$	ISLES: 2.5, 5, **10**, 20 READS: 0.625, 1,25, **2.5**, 5
Cardinality ratio	$\|R\| : \|S\|$	**1**, 2 ,3, 4

Table 2: Experimental parameters (default values in bold).

(5-64), and the READS collection of 5M reads obtained from a human genome with a 5 symbols dictionary and a uniform length distribution (around 100 symbols per string). Both collections were provided by [31].

To perform the experiments, each of the collections with synthetically generated scores is split into two equi-sized (disjoint) partitions denoted by R and S. This way, we avoid performing a self-join which will produce result pairs involving exactly the same object.

To generate scores, we analyzed the real data making two observations. First, object scores usually follow a normal distribution, as shown in Figure 9(a) for hotels in Paris from TripAdvisor.com. Second, a correlation between join attributes and scores may exist. In Figure 9(b), the rating of a hotel is denoted by how dark its red marker is. We observe that the majority of the highly rated hotels are close to each other, and conveniently located next to one of Paris' landmarks, the River Seine. Under these, we distinguish between `score` of type IND and CORR similar to [33]. For IND, `score` values are normally distributed inside $[0, 1]$ and independent to `att` (join) values. In contrast, for CORR, we first randomly generate $|\Sigma|$ *seeds*, and assign them a score uniformly distributed inside $[0, 0.8]$. The generated objects are divided into $|\Sigma|$ clusters based on their distance to the seeds and the score of each object equals the score of its closest seed plus a noise normally distributed inside $[0, 0.2]$.

Finally, to assess the performance of each evaluation paradigm, we measure their response time for aggregate score function $\gamma = SUM$[3], including any index building and/or sorting costs. We measured the performance of the paradigms varying three evaluation parameters: the first is the selectivity of the join, captured by distance thresholds ϵ_{SD} for k-SDJoin and ϵ_{SS} for k-SSJoin. The second evaluation parameter is the number of returning pairs k and the third is the number of seeds $|\Sigma|$ in case of CORR and the synthetic collections. We also perform a scalability and a cardinality test over subsets of the initial synthetic collections varying the $|R| + |S|$ and $|R| : |S|$ parameters. Table 2 summarizes all the parameters involved in our experimental study. On each experiment,

[3]Please note that our analysis can be directly extended to any monotone aggregate function γ.

we vary one of ϵ_{SD}, ϵ_{SS}, k, $|\Sigma|$, and $|R|+|S|$ while we keep the remaining parameters fixed to their default values. The values for ϵ_{SS} are the ones considered also in the String Similarity Search and Join Competition [31]. Further, note that as the value of $|\Sigma|$ increases the score generator tends to generate more independent and less correlated score distributions; for $|\Sigma| = 100$, the generated scores can be considered as uniformly distributed. Finally, also note that both the collections and the indexing structures used by the evaluation paradigms are stored in main memory. In Section 8.7, we discuss how our analysis can be extended in case all involved data do not entirely fit inside the available main memory.

8.2 Focus on Computational Cost

First, we justify our decision to focus on CPU efficient k-Join evaluation for complex data types, instead of minimizing the object accesses from the inputs, similar to previous studies . Assume that inputs R, S reside on disk, already sorted in decreasing order of `score`, and SFP gradually accesses their objects. To calculate the number of I/Os by SFP, consider a 16KB page (typical for modern database systems) and the case of performing only random accesses. To measure the access cost of SFP, we charge 10ms for each page access (similar to a 7200rpm HDD). Table 3 clearly shows that CPU cost overshadows access cost, being up to an order of magnitude higher; an exception arises if the total number of accessed objects is small, e.g., FLICKR with IND scores. Note that in practice access cost can be even less important as some pages are sequentially accessed (e.g. data prefetching) or modern hardware (e.g., SSDs) is used.

collection	score type	number of I/Os	access cost (seconds)	CPU cost (seconds)
BHOTELS-RESTS	–	44	0.44	1.48
FLICKR	IND	2	0.02	0.02
	CORR	128	1.28	3.13
ISLES	IND	9	0.09	0.19
	CORR	704	7.04	29.16
BHOTELS-THOTELS	–	56	0.56	2.39
CITIES	IND	16	0.16	0.75
	CORR	28	0.28	1.27
READS	IND	3,637	36.37	109.41
	CORR	774	7.74	18.39

Table 3: Access and CPU cost of SFP (for default k, ϵ_{SD}, ϵ_{SS}).

8.3 Selecting Index Structure for k-SDJoin

As discussed in Section 5, although R-tree is the dominant indexing structure for spatial data, SFP, JFP and BLP all employ the aR-tree for computing k-SDJoin. In what follows, we justify this decision by comparing three alternative implementations for SFP and JFP termed:

- the aR-tree, as discussed in Section 5.
- the R-tree, which uses R-trees instead of aR-trees to index the spatial join attribute `att`.
- the No-Index, which performs a scan against the buffered objects in case of SFP, while sorts the objects and applies a spatial plane sweep join technique in case of JFP.

Figure 10 reports the response time of SFP alternatives in case of IND and CORR scoring attributes on ISLES, while varying ϵ, k and $|\Sigma|$. As expected, we observe that the aR-tree alternative outperforms the other two as it is able to prune object pairs in terms of both their spatial distance and their aggregate scores. We also observe that the behaviour of the R-tree alternative differs when increasing ϵ_{SD} on IND or CORR scoring types. The reason is the following. As ϵ_{SD} increases, more object pairs qualify the spatial predicate. Since the objects are sorted in descending order of their scores, a smaller number of pairs needs to be examined. However, the increase

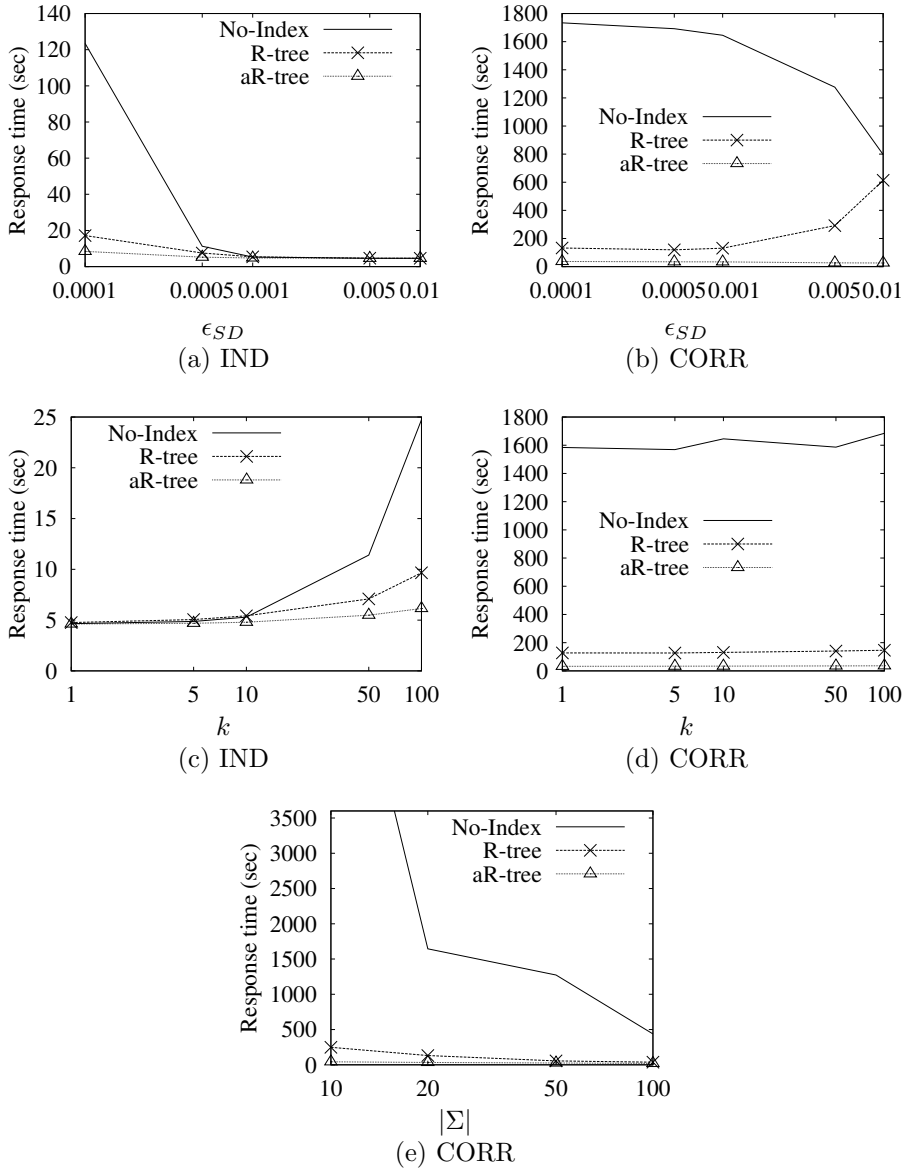


Figure 10: Comparison of SFP alternatives: scoring attributes of type IND & CORR and ISLES.

of ϵ_{SD} also incurs a higher cost for the range queries performed by the R-tree alternative. The effect of this cost is more obvious in case of CORR scoring attributes compared to IND because an overall larger number of object pairs need to be examined. Another important observation is that the response time of the aR-tree alternative is less affected by the varying parameters, due to its ability to prune more object pairs in many circumstances. Finally, with the usage of more seed nodes for score generation, the response time of all alternatives decreases since more object pairs have high aggregate scores.

Figure 11 reports the response time for a similar test on JFP. The aR-tree implementation always outperforms the other alternatives, in some cases for more than two orders of magnitude. This is due to the fact that the R-tree and No-Index alternatives primarily focus on the spatial predicate of k-SDJoin, which is naturally independent of the scores, and therefore, they cannot take advantage of the θ and T bounds as we discuss for the aR-tree alternative in Section 5.2. On the contrary, due to its ability to use score aggregation bounds, the aR-tree based implementation not only outperforms the other alternatives, but it is also very little affected by the increase of ϵ_{SD}. In fact, the overall cost of the aR-tree implementation is dominated by the indexing time

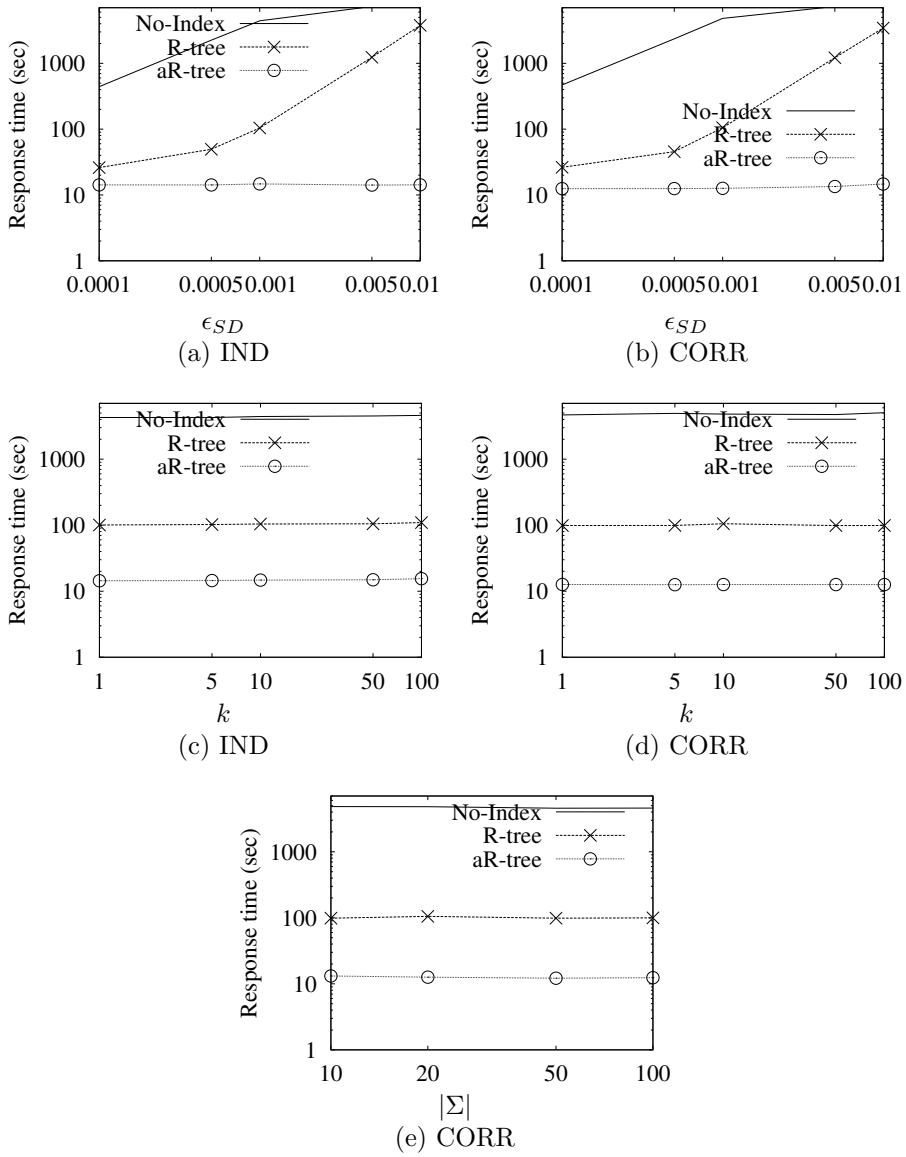


Figure 11: Comparison of JFP alternatives: scoring attributes of type IND & CORR and ISLES (Log-Scaled).

(over 95%). We also observe that all JFP alternatives perform similar on IND and CORR scoring types and are oblivious to the number of seed points $|\Sigma|$; this is because the execution time of the aR-tree alternative is dominated by indexing cost while the other alternatives focus on the spatial join attribute.

Finally, we also implemented BLP with different versions of its block-based join module and the results are reported in Figure 12. They confirm that the bounding and pruning advantages of aR-tree based module is superior to the other alternatives, even when the join is being performed on a smaller block level.

8.4 Estimating any-k and top-k Depths

We first investigate the effectiveness of our model on estimating any-k and top-k depths (Section 7.2). Tables 8.4 and 8.4 report the estimated values of c_R, c_S, d_R and d_S for the default values of ϵ_{SD}, ϵ_{SS}, k, $|\Sigma|$ and $|R| + |S|$ and the real and synthetic score-carrying collections, employing our model and the model proposed in [13]. As expected, when objects are assigned

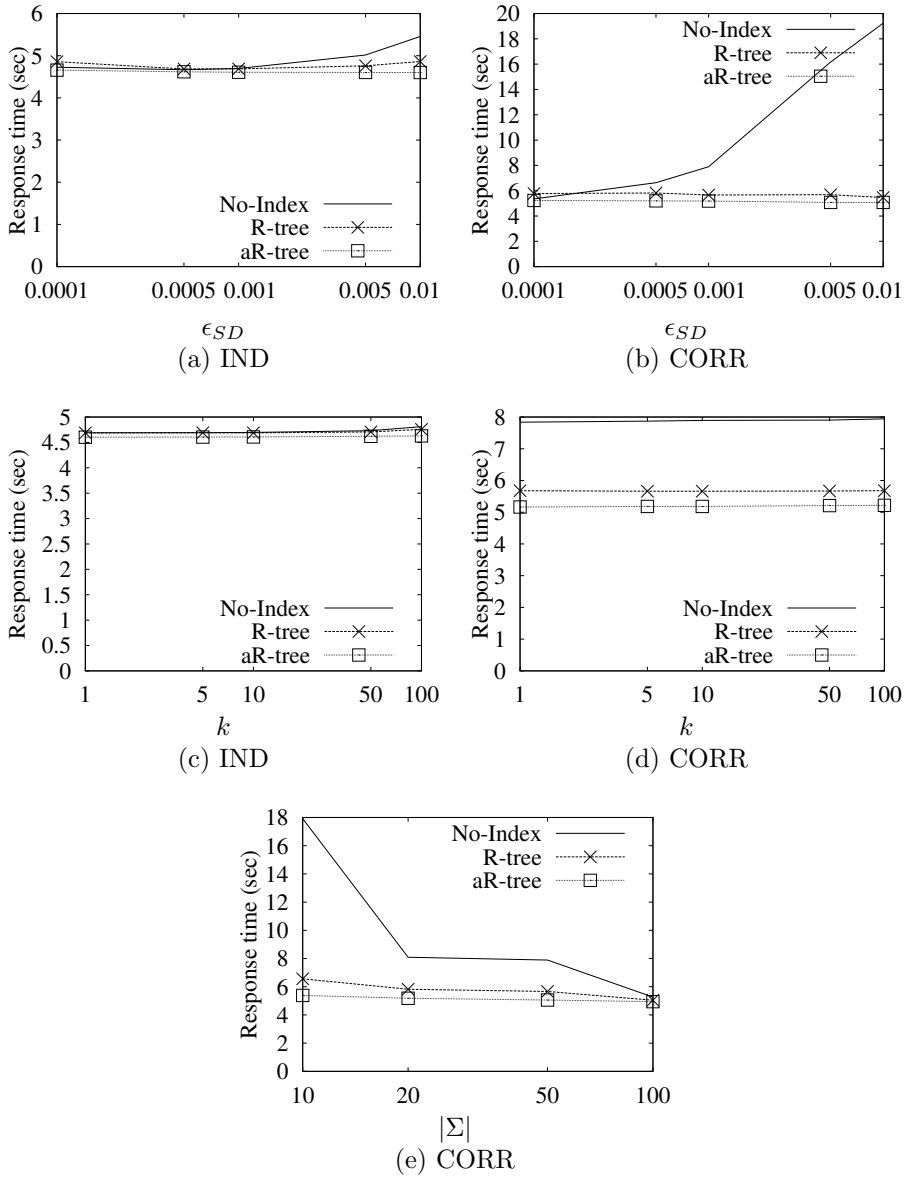


Figure 12: Comparison of BLP alternatives: : scoring attributes of type IND & CORR and ISLES.

IND scoring attributes both models accurately estimate c_R and c_S; the average relative error of the [13] estimation is 18% while of our model is 10%. The values of c_R and c_S are solely related to the join selectivity of the inputs and since the IND scoring attribute is independent of the join attribute, sampling either the upper part of the collections that contains the highly ranked objects (our model), or the entire collections ([13]) both provide a good estimation of the actual join selectivity, i.e., $\sigma_t \approx \sigma$. In contrast, for d_R and d_S, the estimation of [13] model has an 93% relative error over the real values, while our model only 12%. Different from c_R and c_S, the values of d_R and d_S besides the join selectivity are also related to the score distribution. The model of [13] cannot deliver good results unless the scoring attributes follow a uniform distribution and/or prior knowledge regarding the position of the operator in a k-Join hierarchy is available.

In case of CORR type scores, the estimation of [13] model is at least one order of magnitude off the real value of c_R, c_S, d_R and d_S for the FLICKR and ISLES collections; note that, particularly for ISLES, the estimated values of c_R,c_S and d_R,d_S are 3 orders of magnitude smaller than the real values. For CITIES and READS and only for c_R and c_S, the [13] model provides better estimation compared to the other collections, solely because by construction the correlation between

collection	score type	depth	real values	[13]	our model
BHOTELS-RESTS	–	c_R	28,461	17,454	24,317
		c_S	34,215	17,454	24,317
FLICKR	IND	c_R	96	95	99
	IND	c_S	107	95	99
	CORR	c_R	62,969	96	53,352
	CORR	c_S	63,121	96	53,352
ISLES	IND	c_R	322	262	435
	IND	c_S	322	262	435
	CORR	c_R	348,295	262	300,038
	CORR	c_S	348,295	262	300,038
BHOTELS-THOTELS	–	c_R	22,691	1,200	22,378
		c_S	27,131	1,200	22,378
CITIES	IND	c_R	302	273	293
	IND	c_S	322	273	293
	CORR	c_R	1,221	270	1,185
	CORR	c_S	1,172	270	1,185
READS	IND	c_R	7,434	6,625	6,060
	IND	c_S	6,594	6,625	6,060
	CORR	c_R	9,257	6,462	9,420
	CORR	c_S	10,199	6,462	9,420

Table 4: Estimation of any-k depths c_R and c_S.

collection	score type	depth	real values	[13]	our model
BHOTELS-RESTS	–	d_R	27,886	34,908	24,317
		d_S	50,271	34,908	50,271
FLICKR	IND	d_R	1,262	191	1,241
	IND	d_S	1,262	191	1,277
	CORR	d_R	87,582	192	76,962
	CORR	d_S	86,395	192	77,896
ISLES	IND	d_R	5,050	523	5,653
	IND	d_S	6,319	523	7,208
	CORR	d_R	465,531	523	468,544
	CORR	d_S	495,041	523	470,963
BHOTELS-THOTELS	–	d_R	40,733	2,400	41,206
		d_S	49,231	2,400	43,615
CITIES	IND	d_R	6,146	546	5,482
	IND	d_S	6,240	546	5,319
	CORR	d_R	10,057	541	10,057
	CORR	d_S	11,021	541	9,447
READS	IND	d_R	266,878	13,250	222,411
	IND	d_S	256,850	13,250	215,141
	CORR	d_R	57,042	12,924	51,778
	CORR	d_S	54,394	12,924	50,481

Table 5: Estimation of top-k depths d_R and d_S.

the scoring and the string join attribute is not as strong as in case of spatial join attributes. Nevertheless, the model of [13] is always less accurate than ours. On average, the relative error introduced by [13] in case of CORR is 100% for c_R, c_S and 100% for d_R, d_S, while by our model is only 10% and 9%, respectively. Different from IND, when objects are assigned CORR scores a larger part of the collections is accessed to compute a k-Join query since similar objects have similar individual scores which also results in high aggregate scores. The model in [13] is unable to capture this behavior and, hence, the estimated values of any-k and top-k depths are almost identical for both IND and CORR.

Lastly, for the real datasets, the model of [13] always performs worse than ours, due to correlation property of real-world ratings as we discussed in 8.1. Specifically, for BHOTELS-RESTS, the relative error introduced by [13] is 44% for c_R, c_S and 28% for d_R, d_S, while by our model is only 22% and 6%, respectively. For BHOTELS-THOTELS, the model of [13] has a relative error of 95% for c_R, c_S and 95% for d_R, d_S, while our model has only 9% and 6%, respectively.

8.5 Estimating the Optimal Block Size

We first discuss how the procedure of selecting the block size λ for BLP is applied in case of spatial and string join attributes. Recall from Section 7.1 that we model this procedure as an optimization problem, i.e., the optimal value of λ minimizes the objective cost function $\mathcal{C}(\lambda)$:

$$\mathcal{C}(\lambda) = |N_{index}(\lambda)| \cdot \mathcal{C}_{index}(\lambda) + |N_{join}(\lambda)| \cdot \mathcal{C}_{join}(\lambda)$$

The indexing ($\mathcal{C}_{index}(\lambda)$) and joining ($\mathcal{C}_{join}(\lambda)$) costs for two blocks are determined according to the type of the join attribute and the methodology used to perform a block-level join. Specifically, for the two cases studied in this analysis we have:

(i) *Spatial join attributes and* k-SDJoin. We perform an aR-tree based block-level join adapting the R-tree join of [3]. Our experiments in [27] showed the advantage of this approach over a plane-sweep based or an R-tree based join method. The cost of indexing, which involves bulk-loading an aR-tree for each block, is dominated by the cost of sorting the two blocks, i.e., $\mathcal{C}_{index}(\lambda) = \alpha_1 \cdot \lambda \cdot \log \lambda + \alpha_2$, while the joining cost for two aR-trees is linear to the block size, $\mathcal{C}_{join}(\lambda) = \alpha_3 \cdot \lambda + \alpha_4$.

(ii) *String join attributes and* k-SSJoin. We employ the Pass-Join algorithm to join two blocks. Indexing involves partitioning the string attribute of the objects inside the block from collection R and building an inverted index based on a hash-table with a time complexity of $O(\lambda \cdot (\epsilon_{SS} + 1)) = O(\lambda)$ which gives $\mathcal{C}_{index}(\lambda) = \alpha_1 \cdot \lambda + \alpha_2$. For each object in collection S, the joining cost is dominated by the substring selection and the verification step which, according to [18], is also linear to the block size, i.e., $\mathcal{C}_{join}(\lambda) = \alpha_3 \cdot \lambda + \alpha_4$.

Before employing the above cost model to estimate the optimal λ, we first need to approximate α_1, α_2, α_3 and α_4 constants for each k-Join type. To this end, we executed a series of experiments varying the value of λ, and then, employed regression analysis over the collected $(\lambda, \mathcal{C}_{index}(\lambda))$ and $(\lambda, \mathcal{C}_{join}(\lambda))$ values.

We now investigate the efficiency of our model for estimating optimal block size λ. We run BLP for a number of λ values inside the $[1, max\{d_r, d_S\}]$ interval for every combination of join and scoring attribute type. Figures 13, 14 and 15 report the response time of BLP. To make the figures readable and clear we only show the results around the optimal value λ_{opt} of block size and mark value λ_{est} estimated by our model. Our experiments reveal the trade-off between the response time of BLP and the value of λ. Recall that for $\lambda = 1$ BLP operates exactly as SFP but as the block size increases towards λ_{opt} the paradigm increasingly benefits from the block-wise evaluation. However, when λ increases beyond optimal value λ_{opt}, BLP becomes less efficient as it resembles an improved version of JFP which computes an increasing larger part of $R \bowtie_\phi S$. Although our model is not able to find the exact λ_{opt}, the figures show that employing λ_{est} the execution time of BLP for IND and CORR score types, and for the real datasets increases only 3%, 2% and 1% on average, respectively. Note that this estimation procedure is very fast; our experiments show that the time spend to compute λ_{est} corresponds to only the 3.3% of the total response time of BLP, on average. Finally, we also experimented combining the model in [13] for c_R,c_S,d_R and d_S with our model for estimating optimal block size. In this case the average relative increase in the execution time of BLP was 5% for IND but 147% for the real datasets and 260% for CORR, indicating that an accurate estimation of depths is crucial for selecting a good value of λ.

8.6 Comparison of the Evaluation Paradigms

We next compare the Block-based Paradigm against the Score-First and the Join-First for spatial and string join attributes. Figures 16 and 18 report the response time of each evaluation paradigm

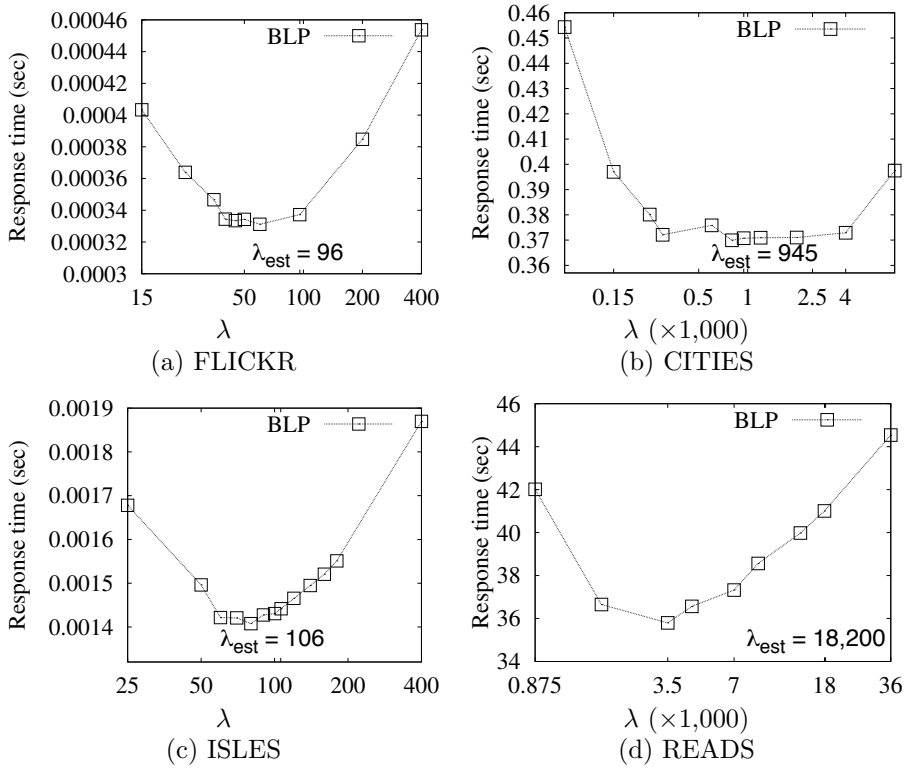


Figure 13: Response time of BLP varying λ: scoring attributes of type IND.

for k-SDJoin and k-SSJoin, respectively, in case of IND typed scoring attributes while reducing the join selectivity, i.e., increasing the ϵ_{SD}, ϵ_{SS} values and varying the number of results k. Similarly, Figures 17 and 19 report the response time of the paradigms in case of CORR scoring attributes while also varying the number of seeds $|\Sigma|$. Note that parameter $|R| + |S|$ is fixed to its default value in Table 2. Our experiments show that BLP outperforms both SFP and JFP in all cases. The advantage of BLP is as expected more significant in case of CORR scoring attributes compared to IND since a larger part of the collections is accessed for the k-Join query. It is also important to notice that BLP is more robust to the variation of the involved parameters compared to the other paradigms.

Regarding parameter k, we observe that all three paradigms are negatively affected by its increase as they examine and compute the aggregate score for a larger number of object pairs, that qualify the join attribute. In other words, the paradigms compute a larger part of $R \bowtie_\phi S$, and this is why the effect of increasing k is more obvious in case of k-SSJoin as string join is more expensive than the spatial join. In contrast, the evaluation paradigms are positively affected by the increase of the number of seeds $|\Sigma|$ for CORR typed scores. This is expected because the value of the scoring attribute is less correlated to the join attribute and intuitively, the object collections resemble more to the collections of IND typed scores. Finally, to better understand the behaviour of BLP, SFP and JFP when varying the join selectivity recall that a k-Join query intuitively comes as a hybrid of a join and a top-k query. In practice, such a combination introduces an intersecting trade-off. Specifically, while increasing either of the ϵ_{SD} and ϵ_{SS} parameters, the join component of the query becomes less selective and thus, more expensive. Yet, as more object pairs qualify the join predicate, the best k results can be now identified faster, sometimes even among the highly ranked objects. In other words, the top-k component of the query becomes less expensive. In the following, we elaborate more on each type of k-Join and discuss in detail how varying the query selectivity affects the response time of the evaluation paradigms.

Our experiments on spatial join attributes are consistent with our analysis in [27]. Still, there exist two important differences. First, in this paper we also experiment with IND typed scoring attributes, and second, we do not set the block size for BLP by hand; instead we employ the

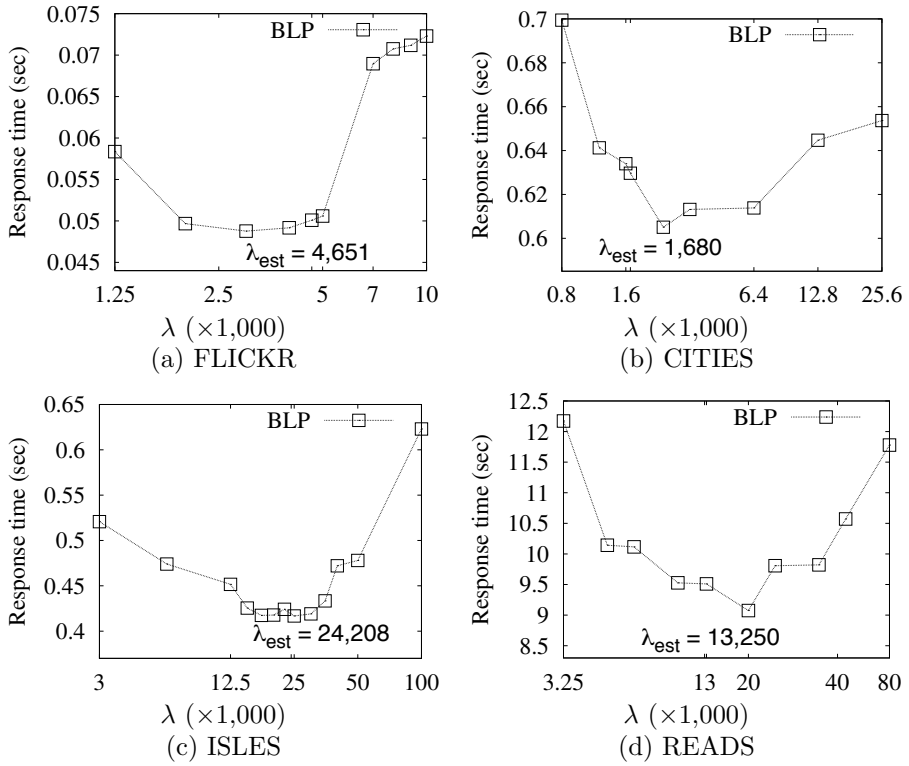


Figure 14: Response time of BLP varying λ: scoring attributes of type CORR.

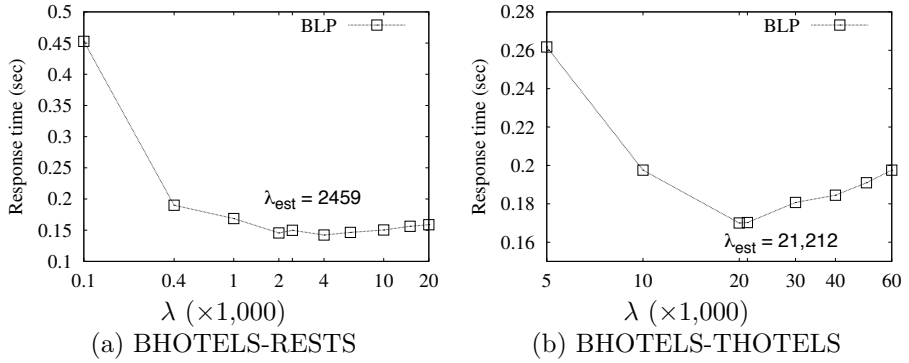


Figure 15: Response time of BLP varying λ: real object collections.

model presented in Section 7.1 and evaluated in Section 8.5. We observe that BLP is always the most efficient approach for k-SDJoin while being very robust to the variation of the ϵ_{SD}, k, $|\Sigma|$ parameters. Due to examining the objects in decreasing order of their score, both BLP and SFP are positively affected by the decrease of the join selectivity, i.e., the increase of ϵ_{SD}, although the benefit is more significant for SFP. It is also important to notice that SFP is faster than JFP for IND scoring attributes but slower for CORR. This is because the insert and update strategy employed by SFP to build the aR-trees over the already examined objects is in practice slower than the bulk loading used by JFP and BLP when a large part of the collections is accessed and indexed, i.e., the case of CORR attributes.

In case of strings as join attributes, we primarily observe that the join selectivity affects differently the evaluation paradigms on each collection. Specifically, as ϵ_{SS} increases the response times of SFP and BLP drop for CITIES while it increases in the case of READS. With the READS collection containing more in number and longer in length strings drawn from a very small dictio-

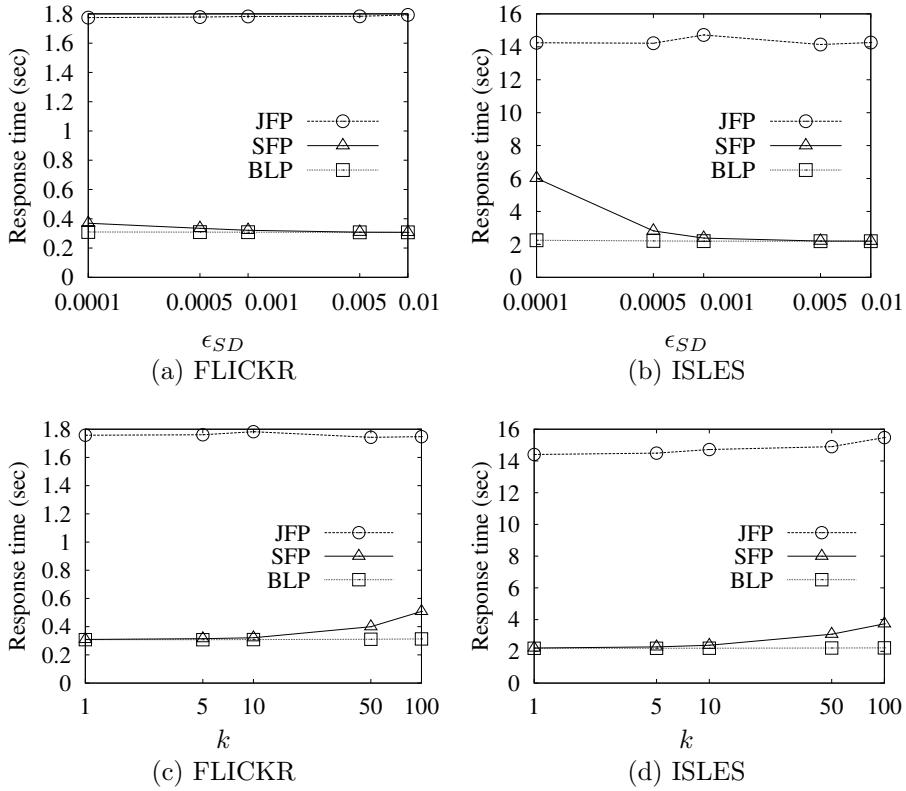


Figure 16: Comparison of evaluation paradigms for k-SDJoin: scoring attributes of type IND.

nary compared to CITIES, the join component of k-SSJoin is far more expensive than the top-k and becomes naturally even more expensive as ϵ_{SS} increases. In addition, due to the nature of the string join, we also notice that JFP can be efficient only for small values of ϵ_{SS} (0 and 1 for CITIES and 0, 4 and 8 for READS) where the execution time of the join component is low.

We also compare BLP against SFP and JFP on real collections varying the number of results k and the join selectivity (thresholds ϵ_{SD} for BHOTELS-RESTS and ϵ_{SS} for BHOTELS-THOTELS). Figure 20 backs up our previous observations; BLP always outperforms SFP and JFP.

Finally, we perform scalability and cardinality tests varying the $|R| + |S|$ and $|R| : |S|$ parameters, respectively, for the FLICKR and READS based object collections and for both IND and CORR types of scoring attributes. Figure 21 and 22 report the response time of each evaluation paradigm. We observe that BLP always outperforms SFP and JFP with the advantage being more obvious in case of CORR scoring attributes. It is also important to notice that BLP scales always better than JFP and in most of setups also than SFP.

8.7 Discussion

As BLP already operates in a block-wise fashion, three minor changes are required when the available memory is limited.First, indices only for the top blocks of the inputs are stored in memory. These blocks contain the highest scored objects and are the most frequently accessed in practice. The remaining blocks are kept on disk and accessed only if necessary. Second, to select a block size, at least two input blocks must fit inside the available memory. Third, the objective cost function $\mathcal{C}(\lambda)$ is extended to also consider the I/O cost for (i) reading the blocks, (ii) storing their indices, and (iii) joining them. SFP and JFP can be also extended, e.g., using disk-based aR-tree for k-SDJoin. However, since both paradigms use a single large index per input, additional updating and accessing costs may incur.

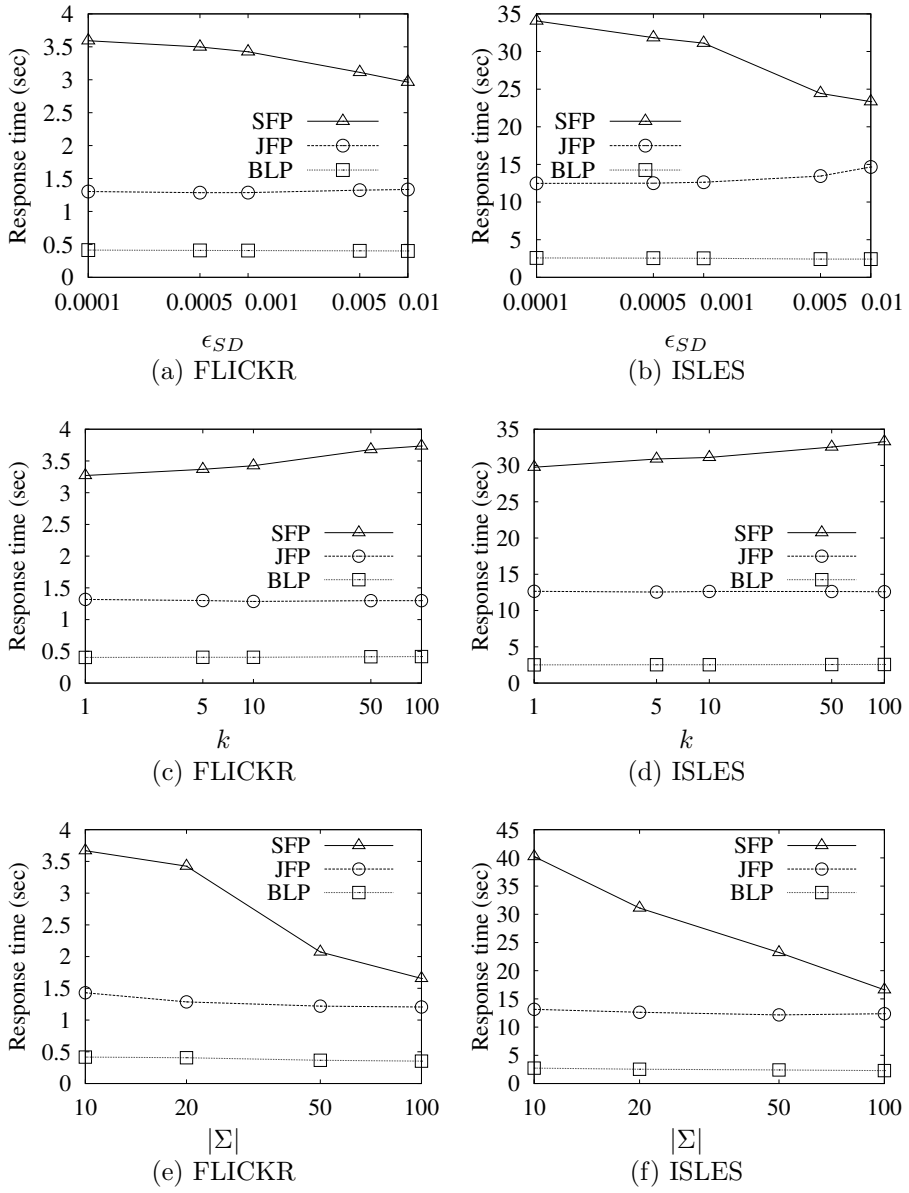


Figure 17: Comparison of evaluation paradigms for k-SDJoin: scoring attributes of type CORR.

9 Conclusions

In this paper, we studied the generalization of top-k joins for non-equijoin predicates on complex data typed join attributes. We first reviewed existing paradigms that can be applied for generalized top-k joins; the Score-First Paradigm (SFP) is a direct generalization of the algorithm in [12], while the intuitive Join-First Paradigm (JFP) primarily deals with the join component of the query. Then, we proposed a novel evaluation paradigm called the Block-based Paradigm (BLP), which applies SFP in a block-wise fashion and joins blocks of the input collections similar to JFP but without computing their entire join. Further, we established the instance optimality of BLP and devised a model for estimating an appropriate block size. As a side contribution, we also proposed a depth estimation model which is much more accurate than the model of [13], because it considers arbitrary score distributions and correlations between the score and join attributes. Next, we showed how the three paradigms can be applied for the cases of spatial joins and string joins between object collections and proposed special optimizations that greatly improve the efficiency of the paradigms. We conducted experiments using real and synthetic data which confirm (i) the

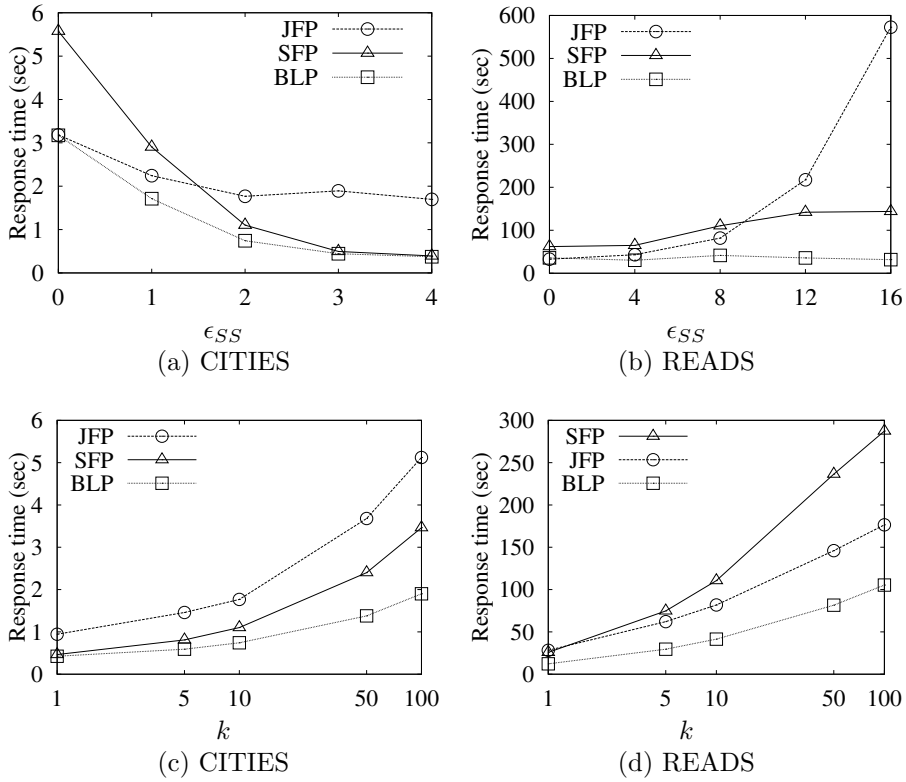


Figure 18: Comparison of evaluation paradigms for k-SSJoin: scoring attributes of type IND.

accuracy of our estimation models for the depth of top-k search and the optimal value of the block size in BLP, and (ii) the superiority of BLP over SFP and JFP under a wide range of parameter settings. In the future, we plan to study the application of BLP in additional cases of join predicates (e.g., overlap joins in temporal databases [9]) and investigate the efficient computation of k-Join in distributed environments.

References

[1] Arge L, Procopiuc O, Ramaswamy S, Suel T, Vitter JS (1998) Scalable sweeping-based spatial join. In: VLDB, pp 570–581

[2] Böhm C, Braunmüller B, Krebs F, Kriegel HP (2001) Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data. In: SIGMOD, pp 379–388

[3] Brinkhoff T, Kriegel HP, Seeger B (1993) Efficient processing of spatial joins using R-trees. In: SIGMOD, pp 237–246

[4] Chakrabarti K, Chaudhuri S, Ganti V (2011) Interval-based pruning for top-k processing over compressed lists. In: ICDE, pp 709–720

[5] Chan EPF (2003) Buffer queries. IEEE Trans Knowl Data Eng 15(4):895–910

[6] Doulkeridis C, Vlachou A, Kotidis Y, Polyzotis N (2012) Processing of rank joins in highly distributed systems. In: ICDE, pp 606–617

[7] Fagin R, Lotem A, Naor M (2001) Optimal aggregation algorithms for middleware. In: PODS, pp 102–113

[8] Finger J, Polyzotis N (2009) Robust and efficient algorithms for rank join evaluation. In: SIGMOD, pp 415–428

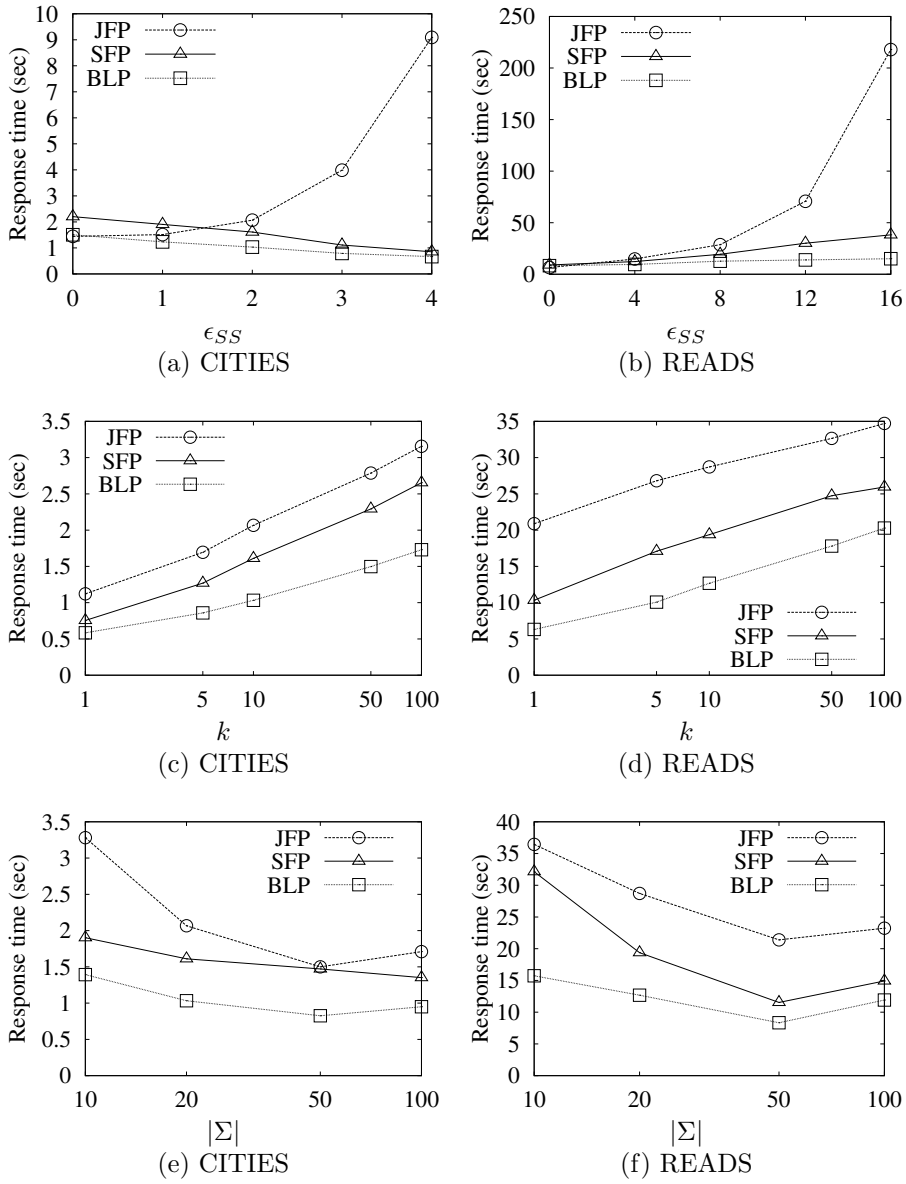


Figure 19: Comparison of evaluation paradigms for k-SSJoin: scoring attributes of type CORR.

[9] Gao D, Jensen CS, Snodgrass RT, Soo MD (2005) Join operations in temporal databases. VLDB J 14(1):2–29

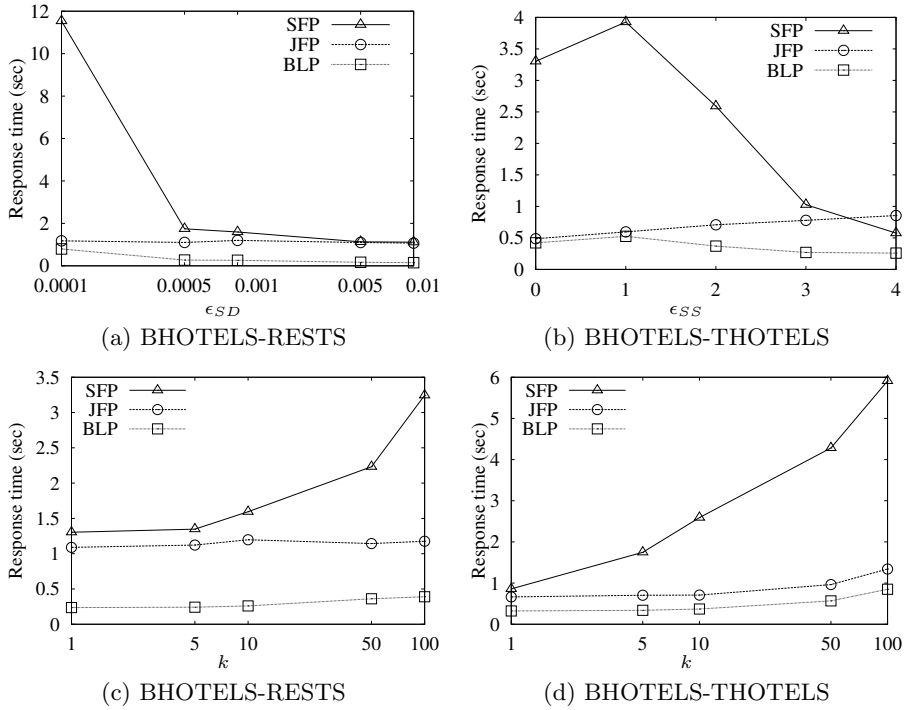
[10] Gravano L, Ipeirotis PG, Jagadish HV, Koudas N, Muthukrishnan S, Srivastava D (2001) Approximate string joins in a database (almost) for free. In: VLDB, pp 491–500

[11] Guttman A (1984) R-trees: A dynamic index structure for spatial searching. In: SIGMOD, pp 47–57

[12] Ilyas IF, Aref WG, Elmagarmid AK (2003) Supporting top-k join queries in relational databases. In: VLDB, pp 754–765

[13] Ilyas IF, Shah R, Aref WG, Vitter JS, Elmagarmid AK (2004) Rank-aware query optimization. In: SIGMOD, pp 203–214

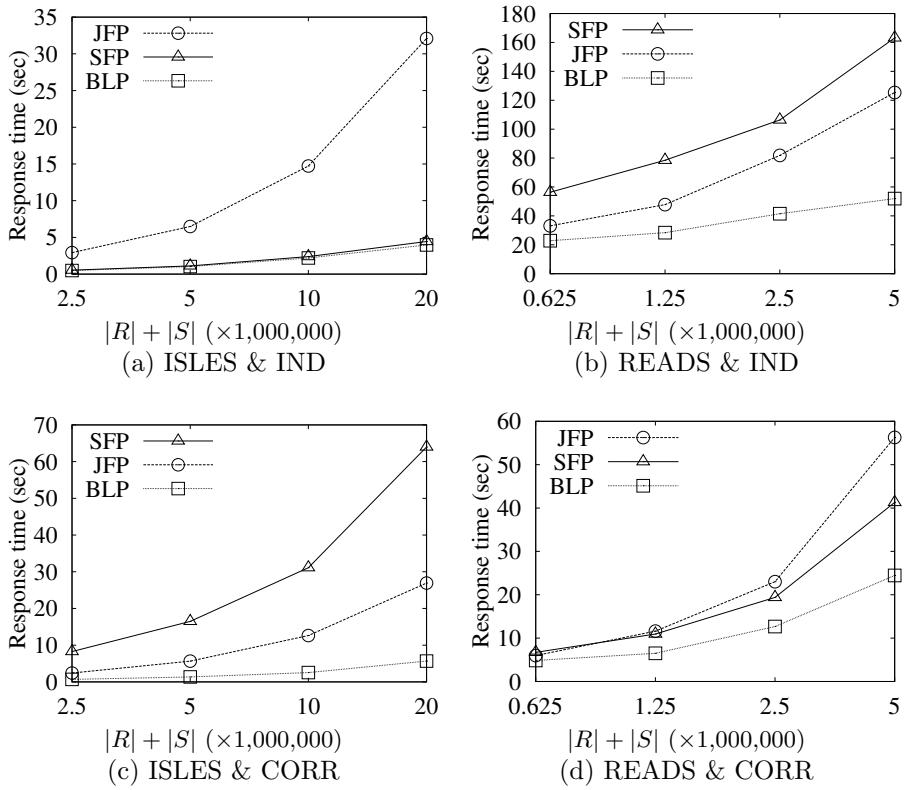
[14] Joachims T, Granka LA, Pan B, Hembrooke H, Radlinski F, Gay G (2007) Evaluating the



(a) BHOTELS-RESTS (b) BHOTELS-THOTELS

(c) BHOTELS-RESTS (d) BHOTELS-THOTELS

Figure 20: Real object collections.



(a) ISLES & IND (b) READS & IND

(c) ISLES & CORR (d) READS & CORR

Figure 21: Scalability tests.

accuracy of implicit feedback from clicks and query reformulations in web search. ACM Trans Inf Syst 25(2)

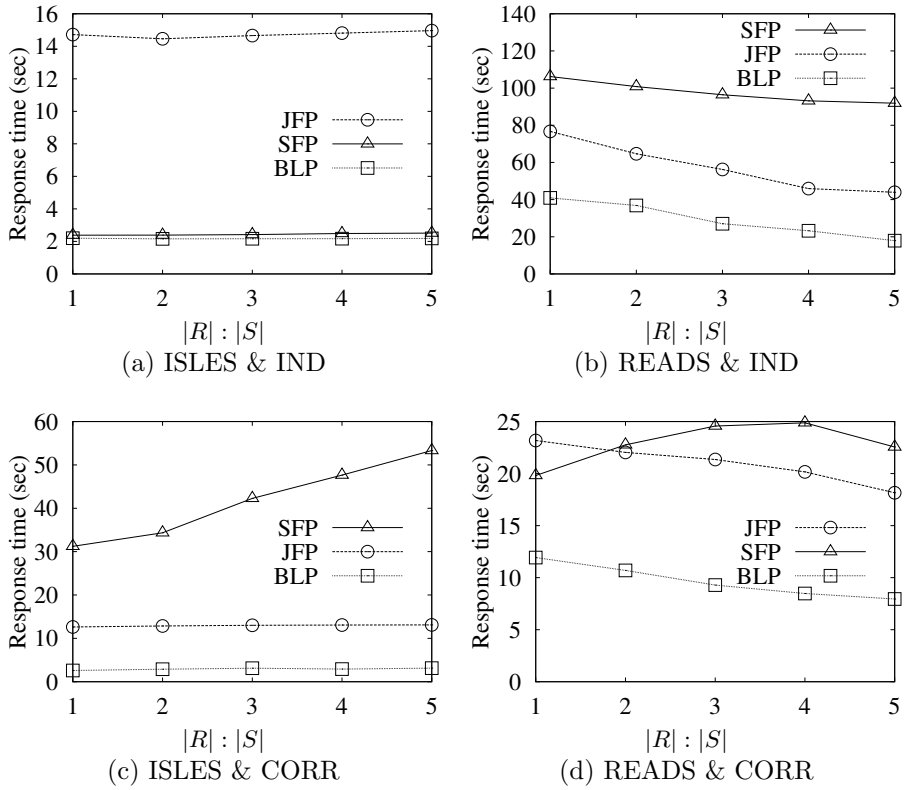


Figure 22: Cardinality tests.

[15] Kiefer J (1953) Sequential minimax search for a maximum. Proceedings of the American Mathematical Society 4(3):502–506

[16] Kim Y, Shim K (2012) Parallel top-k similarity join algorithms using mapreduce. In: ICDE, pp 510–521

[17] Li C, Chang KCC, Ilyas IF, Song S (2005) Ranksql: Query algebra and optimization for relational top-k queries. In: SIGMOD, pp 131–142

[18] Li G, Deng D, Wang J, Feng J (2011) Pass-join: a partition-based method for similarity joins. Proc VLDB Endow 5(3):253–264

[19] Ljosa V, Singh AK (2008) Top-k spatial joins of probabilistic objects. In: ICDE, pp 566–575

[20] Mamoulis N, Yiu ML, Cheng KH, Cheung DW (2007) Efficient top-k aggregation of ranked inputs. ACM Trans Database Syst 32(3)

[21] Martinenghi D, Tagliasacchi M (2010) Proximity rank join. PVLDB 3(1):352–363

[22] Natsev A, Chang YC, Smith JR, Li CS, Vitter JS (2001) Supporting incremental join queries on ranked inputs. In: VLDB, pp 281–290

[23] Ntarmos N, Patlakas I, Triantafillou P (2014) Rank join queries in nosql databases. PVLDB 7(7):493–504

[24] Papadias D, Kalnis P, Zhang J, Tao Y (2001) Efficient OLAP operations in spatial data warehouses. In: SSTD, pp 443–459

[25] Petersen SB, Neves-Petersen MT, Henriksen SB, Mortensen RJ, Geertz-Hansen HM (2012) Scale-free behaviour of amino acid pair interactions in folded proteins. PLoS ONE 7(7)

[26] Poosala V, Haas PJ, Ioannidis YE, Shekita EJ (1996) Improved histograms for selectivity estimation of range predicates. In: SIGMOD, pp 294–305

[27] Qi S, Bouros P, Mamoulis N (2013) Efficient top-k spatial distance joins. In: SSTD, pp 1–18

[28] Sarawagi S, Kirpal A (2004) Efficient set joins on similarity predicates. In: SIGMOD, pp 743–754

[29] Schnaitter K, Polyzotis N (2010) Optimal algorithms for evaluating rank joins in database systems. ACM Trans Database Syst 35(1)

[30] Schnaitter K, Spiegel J, Polyzotis N (2007) Depth estimation for ranking query optimization. In: VLDB, pp 902–913

[31] Wandelt S, Deng D, Gerdjikov S, Mishra S, Mitankin P, Patil M, Siragusa E, Tiskin A, Wang W, Wang J, Leser U (2014) State-of-the-art in string similarity search and join. SIGMOD Record 43(1):64–76

[32] Wang J, Feng J, Li G (2010) Trie-join: efficient trie-based string similarity joins with edit-distance constraints. Proc VLDB Endow 3(1-2):1219–1230

[33] Wu M, Berti-Équille L, Marian A, Procopiuc CM, Srivastava D (2010) Processing top-k join queries. Proc VLDB Endow 3(1-2):860–870

[34] Xiao C, Wang W, Lin X (2008) Ed-join: an efficient algorithm for similarity joins with edit distance constraints. Proc VLDB Endow 1(1):933–944

[35] Xiao C, Wang W, Lin X, Shang H (2009) Top-k set similarity joins. In: ICDE, pp 916–927

[36] Xin D, Han J, Chang KC (2007) Progressive and selective merge: computing top-k with ad-hoc ranking functions. In: SIGMOD, pp 103–114

[37] Zhao K, Zhou S, Tan KL, Zhou A (2005) Supporting ranked join in peer-to-peer networks. In: DEXA Workshops, pp 796–800

[38] Zhu M, Papadias D, Lee DL, Zhang J (2005) Top-k spatial joins. IEEE Trans Knowl Data Eng 17(4):567–579